USENIX

Seventh USENIX Security Symposium Proceedings

San Antonio, Texas *January 1998*

conference

proceedings

Seventh USENIX Security Symposium Proceedings

San Antonio, Texas
January 26–29, 1998

Sponsored by
The USENIX Association in cooperation with the
Computer Emergency Response Team (CERT)

USENIX®
The Advanced Computing
Systems Association

For additional copies of these proceedings contact:

USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA 94710 USA
Phone: 510 528 8649
FAX: 510 548 5738
Email: office@usenix.org
WWW URL: http://www.usenix.org

The price is $27 for members and $35 for nonmembers.
Outside the U.S.A. and Canada, please add
$11 per copy for postage (via air printed matter).

Past USENIX Security Proceedings

Security VI	July 1996	San Jose, California	$27/35
Security V	June 1995	Salt Lake City, Utah	$27/35
Security IV	October 1993	Santa Clara, CA	$15/20
Secuirty III	September 1992	Baltimore, MD	$30/39
Secuirty II	August 1990	Portland, OR	$13/16
Security	August 1988	Portland, OR	$7/7

ISBN 1-880446-92-8

Printed in the United States of America on 50% recycled paper, 10-15% post consumer waste.

USENIX Association

Proceedings of the

Seventh USENIX Security Symposium

January 26-29, 1998
San Antonio, Texas

Conference Organizers

Program Chair

Avi Rubin, *AT&T Labs – Research*

Program Committee

Carlisle Adams, *Entrust Technologies*
Dave Balenson, *Trusted Information Systems*
Steve Bellovin, *AT&T Labs – Research*
Dan Boneh, *Stanford University*
Diane Coe, *Concept Five Technologies*
Ed Felten, *Princeton University*
Li Gong, *JavaSoft*
Peter Honeyman, *CITI, University of Michigan*
Hugo Krawczyk, *Technion*
Jack Lacy, *AT&T Labs – Research*
Hilarie Orman, *DARPA/ITO*
Mike Reiter, *AT&T Labs – Research*
David Wagner, *University of California, Berkeley*

Readers

Katherine T. Fithen, *CERT*
Trent Jaeger, *IBM Watson Labs*

Invited Talks Coordinator

Greg Rose, *QUALCOMM Australia*

The USENIX Association Staff

External Reviewers

Charles J. Antonelli
Dirk Balfanz
Matt Blaze
Daniel Bleichenbacher
Dennis K. Branstad
Ran Canetti
Pau-Chen Cheng
Drew Dean
Shai Halevi
John Ioannidis
Dave Kormann
Kevin Lai
John P. Linderman
David P. Maher
Petros Maniatis
Patrick McDaniel
John Mitchell
Fernando C N Pereira
Benny Pinkas
Tal Rabin
S. Rajagopalan
Jim Rees
Pankaj Rohatgi
Greg Rose
David Safford
Rich Salz
Douglas Lee Schales
Richard Schroeppel
Alan T. Sherman
Michael Stolarchuck
Martin Strauss
Matthew Undy
Dan Wallach
Michael Wiener
Robert Zuccherato

Table of Contents

Seventh USENIX Security Symposium

January 26-29, 1998
San Antonio, Texas

Wednesday, January 28

Thursday, January 29

Distributed Systems

Session Chair: Hilarie Orman, DARPA/ITO

World Wide Web Security

Session Chair: Diane Coe, Concept5 Technologies

Cryptography

Session Chair: Carlisle Adams, Nortel

Preface

In the last year, the popular press has become enamored with computer security. Hardly a week goes by without an article in the *New York Times* or the *Wall Street Journal* about how susceptible the Internet is to hacking, or how someone has broken into some "highly secure" site. All of this press presents exciting opportunities for researchers in the fields of computer security and cryptography, as funding opportunities abound. More and more companies are realizing the importance of securing their networks, their data and their computers. The long-awaited electronic commerce is becoming a reality, and people are actually making money on the Net.

However, for some reason, more and more insecure systems are deployed, giving the press more to feed on. It is a positive feedback loop. The 7th USENIX Security Symposium is a place where the latest research in practical system security is presented and discussed. It is our place to try to break the loop.

As I look over the 65 submissions we received this year (an all-time high), I notice several trends. The topic of greatest interest is the security of mobile code. The advent of Java and other platform-independent languages has enabled computational models that only existed in theory to be rapidly developed and deployed. The inherent security risks have jump-started researchers into action.

There is also a renewed interest in intrusion detection, as new systems far superior to their predecesors are implemented. Several such systems are presented this year. We also have some papers on the security of the World Wide Web.

As you can see, this is a vibrant, dynamic time for our field. It is my hope that this conference and these proceedings will enhance the resources in our community.

Aviel D. Rubin
Program Chair

A Comparison of Methods for Implementing Adaptive Security Policies

Michael Carney
Secure Computing Corporation
2675 Long Lake Road
Roseville, Minnesota 55113
e-mail: carney@securecomputing.com

Brian Loe
Secure Computing Corporation
2675 Long Lake Road
Roseville, Minnesota 55113
e-mail: loe@securecomputing.com

Abstract

The security policies for computing resources must match the security policies of the organizations that use them; therefore, computer security policies must be *adaptive* to meet the changing security environment of their user-base. This paper presents four methods for implementing adaptive security policies for architectures which separate the definition of the policy in a Security Server from the enforcement which is done by the kernel. The four methods discussed include

- reloading a new security database for the Security Server,
- expanding the state and security database of the Security Server to include more than one mode of operation,
- implementing another Security Server and handing off control for security computations, and
- implementing multiple, concurrent Security Servers each controlling a subset of processes.

Each of these methods comes with a set of tradeoffs: policy flexibility, functi.)nal flexibility, security, reliability, and performance. This paper evaluates each of the implementations with respect to each of these criteria. Although the methods described in this paper were implemented for the Distributed Trusted Operating System (DTOS) prototype, this paper describes general research, and the conclusions drawn from this work need not be limited to that development platform.[1]

1 Introduction

Real organizations do not have static security policies. Rather, they have dynamic policies that change, either as a matter of course, or to allow them to react to exceptional circumstances. The computing resources of these organizations must reflect the organization's need for security while affording users the flexibility required to operate in a changing environment.

Any implementation of adaptive security presents its own set of advantages and disadvantages. While this paper compares four methods for implementing adaptive security policies, it is important to keep the needs of the organizations in mind in order to adequately compare implementations of adaptive security. Section 2 outlines some of the possible scenarios requiring adaptive security policies and provides a number of examples of adaptive policies that are useful to the later discussion. Section 3 describes the range of possible implementations for adaptive security given the basic security architecture of the DTOS prototype and provides a brief sketch of the implementations discussed in Section 5. Section 4 provides more background on DTOS, which was used to implement each of the four methods described in this paper. Section 5.1 describes the criteria against which implementations of adaptive security may be measured. The final subsections of Section 5 describes in greater detail the four specific implementations researched at Secure Computing Corporation and evaluates each with respect to the criteria from Section 5.1.

[1] This work was supported by Rome Laboratory contracts F30602-95-C-0047 and F30602-96-C-0210. Portions of the DTOS Overview found in Section 4 appeared in [SKTC96].

2 Motivating Examples for Adaptive Security

The first example of adaptive security consists of organizations that need to change their policies at regular intervals. For example, a bank may have one security policy enforced during business hours and another policy enforced after hours. The business hours policy would grant broad sets of permissions to various sets of employees in order complete normal banking transactions; however, a more restrictive policy would be in effect after hours to prevent system users from altering banking data in unintended ways.

Some organizations may need to release sensitive documents at specific times. For commercial organizations it may be a press release of new product information that must not be available from the webserver until a specified time. Military organizations may have similar needs to make information available to allies on a timed-release basis. Conversely, today's commercial partner or military ally may be an tomorrow's adversary, in which case they should not be allowed to receive various forms of information.

Other organizations may need to adapt their security policies based on the tasks performed by the users. For example, in the banking example cited above, some tasks may be critical to perform despite the more restrictive policy enforced after 5:00 PM. High-priority or urgent tasks may need to be granted special permissions to complete ongoing operations despite the general change of policy. Other task-based policies may make use of an assured pipeline, like that proposed by Boebert and Kain [BK85]. Assured pipelines address situations in which a series of tasks must be performed in a particular order and the control flow must be restricted. An adaptive policy might change the set of permissions associated with a single process as it completes a series of operations. As the process completes one operation, the permission set changes to allow the process to complete the next operation but to prevent it from revisiting any objects that it needed for earlier operations. A related security policy would be the Chinese Wall introduced by Brewer and Nash [BN89], which is intended to prevent conflicts of interest in commercial settings. Briefly, under a Chinese Wall security policy a subject may initially be allowed permission to an entire class of objects, but as soon as the subject accesses one element of the class, permissions to access any other object of that class are denied.

Role-based security policies form another class of adaptive security policies. A role is distinguished from a task in that an individual has an on-going need to complete a set of tasks. (See [SC96], [FCK95], and [Hof97].) In commercial settings, roles may be used to enforce separation of duties [CW87]. For example one role may be granted authority to issue purchase orders while another has authority to pay for those purchases. However, for small companies it may be necessary for one individual to perform actions in more than one role, though not necessarily at one time to provide the proper controls and oversight. In military operations it may be necessary for an individual to perform actions in more than one role simultaneously. For example, in the Navy the role of the Watch Officer on a ship may be performed by the Chief Engineer. This person may need to fulfill both roles simultaneously. Similarly, the Command Duty Officer may need to perform actions reserved for the Commanding Officer in times of emergency. Privilege to invoke these dual roles should be reserved for extreme situations.

Multi-level security (MLS) rules as applied in the military and intelligence communities form a final class of examples of security policies that must be adaptive. Adaptive policies may allow either a relaxation or selective hardening of confidentiality restrictions. Under MLS rules all objects are labeled according to the sensitivity of the data they contain (e.g., Top Secret, Secret, Confidential, and Unclassified). By the simple security rule, users and subjects are allowed access to observe objects only if their clearance level is equal to or exceeds the sensitivity of the object (see [BL73]). During an emergency it may be necessary to consolidate levels into two levels: one for Secret and Top Secret files, and another for the remainder. Thus, under the relaxed rules, someone formerly cleared for Secret could access files formerly labeled as Top Secret. For example, military officers may only have clearance to the Secret level, but once their troops are under fire, they may need to access Top Secret information such as the location or capabilities of enemy forces. Conversely, confidentiality rules and other security measures could be "hardened" based on DEFCON alert status or following detection of a possible intrusion. There are a number of ways to "harden" a system. For example, one could increase internal controls, perform full audits rather than selective audits, or require additional authentication measures.

3 Implementation Space

The DTOS prototype provides a security architecture that separates the enforcement of the security policy from its definition. Since this type of security architecture is not unique to the DTOS prototype, results from this paper apply to a variety of systems with similar architectures.

Elements available to adapt the security policy include the following:

- the number or complexity of the databases that a Security Server uses to initialize its internal state
- the number of Security Servers available to the microkernel for security computations
- the control over which Security Server makes security computations on behalf of the microkernel

Although the number of possible implementations is large, this paper describes the following representative implementations:

- One Security Server and multiple databases — adapting the policy by forcing the Security Server to re-initialize from a new security database.
- One Security Server and one database — adapting the policy by expanding the internal state of the Security Server and increasing the complexity of the security database to describe more than one set of security policy rules and by providing the Security Server with a mechanism for changing its mode of operation.
- Multiple Security Servers with a single active server providing one point of control over security computations – adapting the policy by providing a mechanism to hand off the responsibility of computing access decisions from one server to another. Thus, one and only one Security Server defines the policy at any given time.
- Multiple, concurrent Security Servers with responsibility for security computations partitioned by tasks — adapting the policy by assigning a pointer to a specific Security Server to each new process. In this method, whenever a process makes a request to the microkernel for service, the microkernel submits requests for access computations to the Security Server that is associated with that process and which defines the security policy with respect to that process.

4 DTOS Overview

This section provides an overview of DTOS, the operating system used to implement the four methods discussed in this paper.

DTOS was designed around a security architecture that separates enforcement from the definition of the policy that is enforced. This architecture allows the system security policy to be changed without altering the enforcement mechanisms. The policy is defined as a function that maps a pair of security contexts to a set of permissions. Each pair of security contexts represents the security context of a subject and the security context of an object that the subject attempts to access. Currently, DTOS implements security contexts consisting of level, domain, user, and group, but the set of attributes that form a security context is configurable. Enforcement consists of determining whether the permissions specified by the policy are adequate for an access being attempted. The generality of the DTOS security architecture has been studied as part of the DTOS program [Sec97]. The conclusion of this study is that a large variety of security policies, useful for both military and commercial systems, can be implemented.

The basic DTOS design consists of a microkernel and a collection of servers. The microkernel implements several primitive object types and provides InterProcess Communication (IPC), while the servers provide various operating system services such as files, authentication, and a user interface [FM93, Min95]. Of particular interest is a *Security Server* that defines the policy enforced by the microkernel and also possibly by other servers. When a request is made for a service provided by the microkernel, the microkernel sends identifiers for the security contexts of the subject and object to the Security Server. These identifiers are referred to as *security identifiers* or *SID's*. A context contains attributes about a subject or object that are necessary for making security decisions. For example, the context may contain the domain of a subject or the type of an object, or the level of a subject or object. The information that makes up the context is dependent on the policy. The actual contexts are local to the Security Server and are not available to the microkernel. The Security Server then computes permissions for the context pair, as defined by the policy that it represents, and replies to the microkernel. The microkernel is ignorant of the context of each entity since it only enforces the permissions that the ß computes on its behalf. Finally, the microkernel determines if the permissions required for the request were present in the reply. Other servers can communicate with the Security Server in a similar fashion.

For example, a Security Server implementing an MLS policy might maintain subject and object contexts consisting of a level. For the microkernel to enforce the simple security and *-property of the Bell and LaPadula model [BL73], the Security Server will grant a write permission only if the level for the object security identifier dominates that of the level for the subject security identifier, and it will grant read permission only if the level for the subject identifier dominates that for the object identifier. Both permissions may be granted if the levels are equal. A file server would check for write permission before allowing a request to alter a file. An alternative Security Server might provide UNIX-like access controls by maintaining a user and a group for each subject context and an owner, group, and access control bits for each object context. This type of Security Server will grant permissions based on the access control bits depending on whether the user in the subject context matches that of the owner and whether the groups match.

A prototype DTOS microkernel and Security Server has been built by Secure Computing. The microkernel is based on Mach, developed at Carnegie Mellon University [Loe93, Ras91]. A version of the Lites UNIX emulator, modified by the government, provides secured UNIX functionality.

The object types implemented by the microkernel include *task*, *thread*, and *port*. Tasks and threads represent the active subjects, or processes, in the system. Each task has a security context that is used for security decisions involving that task. The state of each task includes virtual memory consisting of a set of disjoint memory regions, each of which is backed by a server that is used to swap pages of the region in and out of physical memory. Each task contains a collection of threads, each of which is a sequential execution, that share the task's virtual memory and other resources. A server is implemented as one or more tasks.

The ports are unidirectional communication channels that the tasks use to pass messages. Tasks use *capabilities* to name ports, and these are kept in an IPC name space on a per task basis. Each capability specifies the right to either receive from or send to a particular port. These capabilities may be transferred to another task by sending a message. For each port there is exactly one receive capability. Therefore, at most one task can receive messages from the port. IPC is asynchronous in that messages are queued in the port and the sending task does not wait until its message has been received. An exception is when the microkernel is the receiving task, in which case the sender waits until the microkernel finishes processing the message.

Sending or receiving a message is a Mach microkernel operation to which DTOS has added security controls that enforce the security policy. Thus, possession of the appropriate capability for a port is necessary but not sufficient in order to send or receive a message from that port. The security contexts of the task and the port must also permit the operation. The policy also constrains what capabilities may be passed in a message sent or received by a task.

The Security Server receives requests from the microkernel through the *microkernel security port* and from other servers through a *general security port*. Requests contain four elements:

- an operation identifier — allowing the Security Server to specify history-based policies that depend on the sequence of operations made on an object,
- a subject security identifier (SSI) — representing the security context of the subject,
- an object security identifier (OSI) — representing the security context of the object, and
- a send capability for a reply port.

The Security Server replies by sending the permissions for that pair to the reply port (Figure 1). Not shown in this figure is the fact that the Security Server both defines and enforces a policy for the requests that it receives. It might allow security determination requests from some subjects, but not from others. Similarly, it might allow security determination requests from a particular subject only for certain (SSI,OSI) pairs.

Security enforcement as described above would be very expensive due to the large number of messages that must be exchanged between the microkernel and the Security Server. The solution in DTOS is

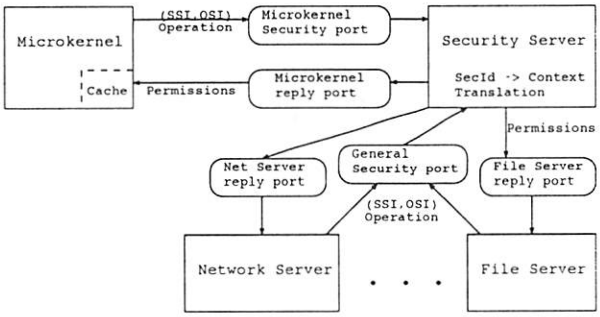


Figure 1: Security Server Interaction

to cache (SSI,OSI) pairs with their permissions in the microkernel [Min95]. When the microkernel receives a request, it first looks in the cache for the appropriate (SSI,OSI) pair. If that pair is in the cache, the microkernel uses the cached entries. Otherwise, it sends the pair to the Security Server to determine the permissions, usually also caching the reply. (Part of the permission set returned is permission to cache the reply — caching would not be permitted for permissions granted for a single operation by a dynamic policy.) Since sending to and receiving from a port are microkernel operations controlled by the policy, the cache must be preloaded with permission for the Security Server to send and receive from the designated ports.

In order to implement a different policy (either by changing the current ß or by referring to a new ß) there must be a mechanism for flushing permissions from the microkernel's cache. Otherwise, if the new policy removes permissions from the system for a specific (SSI, OSI) pair, and the microkernel has already cached the permissions for that pair, then the microkernel would continue to enforce the old policy rather than consult the ß defining the new policy. Therefore, the ß must issue a command to the microkernel, and any other servers registered as caching permissions determined by the ß, telling it to flush its cache. However, it would be impractical for the microkernel to flush every permission in its cache; if it did, then the entire system would come to a halt. Therefore, some permissions are hard coded. These include some of the basic permission required for IPC between the subjects comprising the operating system itself.

The separation between policy and enforcement in the DTOS prototype make it attractive for studying adaptive security. The work described in this report discusses refinements to the design that are important for these policies.

5 Comparison of Implementations

This section of the paper describes each of the methods for changing the security policy in greater detail along with the capabilities and limitations presented by each. However, we begin by describing the criteria against which the four implementation methods are evaluated in Section 5.1. The methods themselves are described in Sections 5.2 through 5.5

5.1 Criteria for Evaluation

An adaptive security policy for a computer system must have the flexibility to meet the security requirements of the organization that fields the system. There are two types of flexibility to consider:

- Policy flexibility — the range of policies that a system can support before and after a transition between policies.
- Functional flexibility — the ability of users to complete tasks despite the transition of policies.

However, greater flexibility may come at the expense of security, and the greater complexity required for some types of transitions may also have an impact on the reliability of the system.

The criteria identified here are not independent of one another; in fact, examining various implementations of adaptive security leads to a series of tradeoffs with respect to these criteria. The conclusions that are drawn from the analysis of the four implementations reflect the nature of the dependence of the criteria upon one another.

Policy Flexibility In the context of adaptive security, the concept of policy flexibility could be measured by the amount of change one is allowed to make and whether the system can enforce an arbitrary new policy. Thus, policy flexibility depends on the number (or lack) of constraints that must be satisfied by the successor policy for a given predecessor policy.

Functional Flexibility Functional flexibility addresses whether the policy transition is graceful or harsh with respect to the applications that are running at the time of the transition. A harsh transition might be like turning off the power and re-booting the system, whereas a graceful transition may appear seemless to the user and most applications on the system. A harsh policy transition may prevent users from performing necessary, possibly urgent, tasks, rather than allowing them to complete their tasks in an evolving security environment. The ideal is to allow necessary tasks to complete while terminating tasks that are not only disallowed under the new policy, but which represent a security risk in the new environment.

Security The existence of a mechanism or method of changing policies may introduce security vulnerabilities. In assessing a method of policy adaptation, one must consider the security risks that are inherent in that method. Furthermore, each type of policy transition must be assessed for the relative difficulty of providing formal assurance evidence in support of the policy transition.

Reliability Each method of policy transition introduces a measure of complexity into the system. Changing policy may expose the system to certain risks which decrease the stability of the entire system.

Performance The ability to change policies quickly has impact on the needs of the user for security, functionality, and reliability. A complex hand-off may allow greater flexibility between policies enforced before and after the transition, but it may also present greater security risks. A less complex hand-off may provide performance gains at the expense of functional grace or flexibility.[2]

5.2 Loading A New Policy Database

One possible method for implementing a new security policy is to change the way that the Security Server defines it by creating a second database and re-initializing the Security Server. A method for doing this existed on the DTOS prototype. During the boot process, the microkernel operates on a hard-coded cache of permissions until the Security Server is ready for operation. Once the Security Server has initialized, the microkernel places the command **SSI_load_security_policy** on the security port of the Security Server. This command causes the Security Server to read the security database to construct a table in its internal memory that maps SSIs and OSIs to permissions. The Security Server then tells the microkernel to flush its cache of permissions, and from that point onward, the policy defined by the Security Server is the policy enforced by the microkernel. The same command can be used to replace one table with another. Once the Security Server has loaded the new policy, it tells the microkernel to flush its cache, and the new policy is enforced by the microkernel.

The command to reload the policy can be encapsulated in a user-invoked program or in some automated process which changes the policy at the triggering of some event. Thus, the policy can be changed at regular intervals using a process like the UNIX utility *cron*, or by a background process which monitors the system for intrusion events.

Policy Flexibility This method relies heavily on the tables that can be loaded into the Security Server from the security database. Since the tables are indexed by the SSI and OSI, the management of the system is easiest if the Security Server loads a new policy which is similar to the old one; thus limiting the granularity of the allowable policy changes. A radical change of policy requires that each entity in the system have a security context which can be recognized by the active Security Server both before and after the policy change.

For initial policies based on Type Enforcement[3] (see [BK85]) or MLS access rules, it would be difficult to make radical changes in the policy. Every entity that has a type or domain associated with it must also have the attributes necessary for enforcing the different policy. Thus, to change from a Type Enforcement policy to a UNIX-like security policy, it would be necessary for objects and processes to have attributes necessary for both security mechanisms. For objects it is necessary to maintain contexts for the sensitivity level of the object as well as the users and groups which may have access to the object. For subjects, it is necessary to maintain the clearance level of the subject as well as the user of the subject. It is also necessary to maintain a database listing the group membership.

[2] Performance seemed to a natural criterion to include. Unfortunately, the performance data is incomplete. Despite this problem, the authors have chosen to include partial data although it is somewhat inconclusive.

[3] A thumbnail sketch of Type Enforcement describes it as a type of mandatory access control policy in which each object has a security attribute known as a *type* and each subject has an attribute known as a *domain*. Subjects are granted access to read, write, or execute objects based on the domain-type pair. Roles can be construc.ed for users by forming sets of domains in which users may have subjects operating.

Functional Flexibility Since the transition between policies during the loading of a new policy is nearly atomic, this implementation is quite harsh on running applications. Any application which ceases to have permission to perform any task under the new rules is essentially orphaned. This abrupt change of behavior is probably acceptable, and may even be desirable in some contexts (e.g., military emergencies). However, in some contexts this abruptness would cause considerable difficulty. In the banking example presented in Section 1, there may be occasions when a particular user must complete a specific transaction before the end of the day. However, if the policy transition time occurs at 5:00 PM sharp and the user needs an additional fifteen to twenty minutes to complete the task, then the policy may hinder bank employees from completing vital tasks. This would doubtlessly be unacceptable under this scenario.

Security The immediacy of the transition of this method provides for the greatest security; the users always know exactly which policy is the current policy. As will be shown below, this is not always the case with other methods.

Although the security database is a critical object that should be protected from unauthorized modification, the security database could be changed while the system is in operational mode. Assuming that the system is fielded with adequate physical and procedural security constraints, the security database is more susceptible to replacement or modification during operation than the database (and system) would be to attacks conducted between successive boots of the system. If subverted software could replace the intended database with a different file, the system would enforce the wrong policy.

The issue of who can authorize, authenticate, and execute the policy change is a clear security concern. In the DTOS prototype, authority to reload the security policy is restricted to subjects that have the permission *ss_gen_load_policy*. Authorization to operate subjects with this permission can be restricted to certain processes, to roles, or to sets of individuals with other security mechanisms.

As for the assurance of such a system, there would be some concerns about the flow of information across transitions. Consider the following example: Suppose that under one policy a subject in domain D_1 has no permission to observe data in objects of type T_1 but does have permission to observe data in objects of type T_2. Furthermore, no domain that may observe objects of type T_1 may write to any object of a type that D_1 may observe (e.g., T_2). Under this policy alone, there is no possible flow of information from type T_1 to domain D_1. Under a second policy, subjects in domain D_2 may observe data in T_1 and write to objects of type T_2, but D_1 is no longer authorized to observe objects of type T_2. Under the second policy, no information may flow from T_1 to D_1. However, after a transition from the second policy to the first, it would be possible for a subject in domain D_2 to pass information from objects of type T_1 to a subject in domain D_1.

Similarly, when dealing with MLS policies and dynamic security lattices, one is necessarily concerned about loss of confidentiality and potential contamination of files during periods of relaxed security. Returning to a more stringent MLS policy or changing a policy using Type Enforcement requires extensive audit to effect such "roll backs." Unfortunately, these concerns exist for all methods.

While there do not appear to be any theoretical frameworks nor any tool support for conducting covert channel analyses (and simialr analyses) for adaptive security policies, some of the formal modeling and proofs might be relatively easier for a policy in which the database is simply reloaded than for more complex policy transitions.

Reliability A tangible concern is that if the database file has become corrupted, then the Security Server will not be able to read it. The effect of this is that the Security Server dies, and the system is left without any Security Server at all. Not only would the system not be able to enforce the new, intended policy, but the system would have difficulty running at all. The microkernel and other processes that can cache permissions computed by the Security Server would rely solely on the permissions that had been cached up to the time that the Security Server went down.

Both the security and reliability concerns could be ameliorated by placing a checksum (or a digital signature) over the security database. The Security Server would not read in the new database unless the checksum can be verified.

Of course the discussion in the previous two paragraphs assumes that the specification of the new database is correct. Even if the policy changes are small, an entirely new database must be constructed, and it must be correct to avoid problems (e.g., either a corrupt file or deadlock conditions between the Security Server and microkernel). Since it can be difficult to specify one database correctly, attempting to make more extensive changes reduces the reliability of this method.

Performance This is the second fastest method for changing policies. During performance testing, a typical transition time (median) required 2.985 seconds, and no transition required more than 3.970 seconds. Although this might not be as fast as necessary in a real-time embedded system, this would be more than satisfactory in systems such as the banking application mentioned in the introduction.

5.3 Expanding the Database and Security Server State

In this method of transition between policies, when the Security Server loads its initial security database, all of the permissions allowed under all modes of operation are initialized in the Security Server's internal memory. A mechanism internal to the Security Server allows it to change policy without having to read a new security database. Thus policy changes could be triggered by a variety of events. The policy could change based on several events: the time of day, when a process completes a certain task or invokes a certain permission, or when an alarm is set off (e.g., by a possible intrusion event). This method is similar to the mechanism described in Section 5.2; however, because of the ability to change policies based on triggering event it has a number of advantages which are listed below.

Policy Flexibility This method has the same restrictions that loading a new database has. It is easiest for the Security Server to alternate among policies which are similar. For initial policies based on Type Enforcement or MLS access rules, the new policy must also be based on Type Enforcement or MLS access rules. However, the mechanisms for changing policy definition give this method greater flexibility than the previous method.

For example, for policies which change on a regular periodic basis (recall the banking example in which a more stringent policy is enforced for after-hours operation), a timing mechanism that triggers the change of policy could be added to the Security Server.

Another adaptation mechanism could be triggered by the use of particular permissions. For example, when a particular permission is requested and returned to the requesting process, that permission could be removed from the Security Server's notion of the allowed permissions. This would render the permission as a one-time only permission. For example, in a commercial application a one-time permission to issue payment for a purchase order would prevent double payment.

Similarly, when a particular permission is requested and returned to the requesting process, that permission could be removed from the Security Server's notion of the allowed permissions, and one or more could be added. Such adaptations could be chained together. For example, if the Security Server were applying Type Enforcement, a process operating in one domain might be granted access to a new type and denied access to an old one. Thus a set of operations could be performed by a single process in a secure pipeline. Such secure pipelines are already possible with Type Enforcement, but each operation is performed by a separate process, each running in a unique domain (see [BK85] and [GHS97] for more details). This type of mechanism would also be ideal for enforcing the security policy known as Chinese Wall (see [Pfl97] for a definition).

Functional Flexibility Since the transition between policies during the loading of new policy is the most atomic, this implementation could be as harsh on running applications as reloading the database. However, the database could be expanded to include several policies so that a policy transition could take place with several intermediate policies during the transition. A phased transition of this sort might allow some tasks to complete processing within fixed time limits.

Security The security concerns here are the same as in Section 5.2 with the exception that the security database is read once and only once at initialization, and thus the possibility that an untrusted user or process has been able to corrupt it is removed from concern.

With the expanded state of the Security Server, changes of policy may be regulated automatically by the time of day, as in the banking example, or by events, as in the Chinese Wall policy. By moving the authority for changing the policy from subjects to events, the methods by which hostile users could alter the enforced policy change. If a hostile user tampers with the system clock, or forces a triggering event, or counterfeits a triggering event, then he could control changes of policy.

The ability to "harden" system defenses automatically in the event of a possible intrusion also seems to be a particular advantage not present in reloading the database.

Reliability This method is more reliable than reloading the policy because we are not concerned about the second policy being corrupted after boottime. However, like the previous method, changes to the policy are limited in their granularity by practical concerns. This method makes the coding of the Security Server more complex, which may cause unforeseen problems; so small policy changes, this method would be superior to specifying and reloading an entirely new database. However, one should not be tempted into making too many changes to the policy using this method because of the potential complexity.

Performance Explicit performance numbers are not available for this method. However, since it avoids the time-consuming step of reading a new database, it is anticipated to be faster than reloading the database, and expected transition times should be less than one second. Thus, it is expected to be the fastest of the four methods under discussion.

The microkernel and other processes can cache permissions to improve performance; so changing policy and flushing the cache frequently could cause a minor performance drag. However, permissions in the database can be flagged as non-cachable. Thus, transient permissions as described above could be flagged in that way so that the microkernel would not have to flush its entire cache as it does for reloading the database. Similarly, permissions in the database can be flagged as those which cannot be flushed. Thus, persistent permissions could be flagged so that the microkernel would not have to flush those permissions from its cache at all, and performance would not be adversely affected by the adaptation of policies.

5.4 Handing Off Control to a New Security Server

In the Security Server hand-off, the current Security Server passes the receive capability for its security port to another Security Server that implements a new policy. In order to accomplish this, the new Security Server is initialized while the current Security Server is still in control of the policy decisions. The new Security Server uses the command **get_special_port** to obtain the send right to client port of the current Security Server and then issues the **transfer_security_ports** to the current server. The current Security Server packages the receive rights for its security port along with two other tables of information. One table contains the mapping between security contexts and SIDs that it uses to interpret incoming requests, and the other table lists the ports of processes that may be caching security permissions. The new Security Server needs the former to interpret requests that it receives regarding any processes or objects that exist prior to the hand-off. It needs the latter because it may eventually need to tell these other processes to flush their cached permissions. The last action of the current Security Server is to tell all processes with cached permissions to flush their caches. At this point the new Security Server can compute access permissions, and the microkernel and any other processes that enforce these permissions can enforce the new security policy.

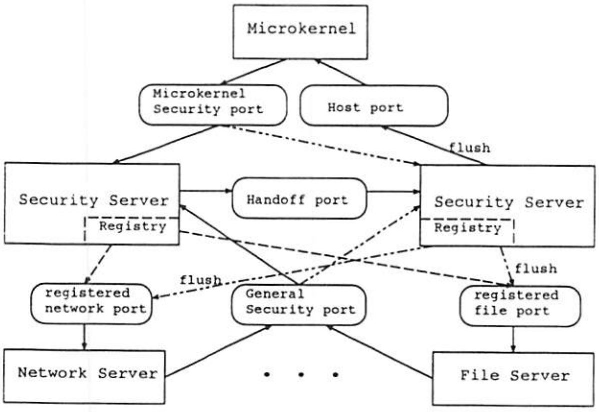


Figure 2: Security Server Hand-Off

In order to be able to process new requests for permission computations, the new Security Server must be able to interpret the requests. As mentioned above, the old Security Server sends the appropriate information for the new Security Server to match contexts to SIDs. However, the new Security Server has some knowledge of security contexts prior to receiving this information from the old Security Server; so it must reconcile its understanding of contexts with the mapping information received from the old Security Server. It also must create new SIDs for any new contexts which were not recognized by the old Security Server. For example, if the both the new Security Server and old Security Server are implementing Type Enforcement and there are new domains as part of the new policy, the new domain must receive a SID. Similarly, if the hand-off occurs in order to implement dynamic lattices as part of an adaptive MLS policy, any new levels must receive SIDs. Once the new Security Server has completed this reconciliation, the old Security Server can shut down.

Policy Flexibility The greatest strength of the hand-off method is that one can enforce a global, radical change of policy. The new Security Server can implement a very different policy from the one that is enforced before the hand-off. Not only can the new Security Server initialize from a new security database, it can implement an entirely different set of algorithms for making its security computations. This may be especially important for implementing dynamic lattices as part of an adaptive MLS policy.

As discussed in Section 5.2, the only impediment to changing the policy in a radical way is the labeling of objects and processes with the appropriate set of attributes which can be interpreted by both the new and old Security Servers. In other words, radically different policies may require essentially disjoint sets of attributes which the system designers glue together for the context of any single entity.

Functional Flexibility In essence this method is not different from reloading the database. Changes to the security policy are global and atomic. The same problems exist in this method as for reloading the databse for situations where a harsh change of policy is undesirable, as in the banking example.

Security Some of the same security advantages and concerns exist here as for the Reload Policy method. As with the Reload Policy method, the users always know exactly which policy is the current policy. However, if the new Security Server has to initialize from some static file or security database, there is always the risk that it could be subverted. Another possibility is that the code for a new Security Server could be subverted as well and that a malicious Security Server could end up in control of the permission decisions.

There remains the issue of who can authorize, authenticate, and execute the policy change. The Security Server will hand off the security port to the new server when it receives the command *SSI_transfer_security_ports* on its security port. Just as in the case of the authority to reload the policy, the permission to issue this command is restricted to subjects that have the permission *ss_gen_load_policy*. Authorization to operate subjects with this permission can be restricted to certain roles or to sets of individuals with other security mechanisms. The additional concern here is that the security port is transferred to the correct subject, the new Security Server.

Reliability The hand-off is a more complicated procedure than the preceding two methods from two points of view: of the operation itself and of the development of the system.

From the latter point of view, the handoff requires that a second, fully functioning Security Server be implemented, as well as a second security database. This solution should not be used for trivial changes to the policy.

Unfortunately, from the former point of view, the hand-off procedure on the DTOS prototype is relatively delicate, and this is its greatest weakness. While these type of problems are inherent in all four methods, they are more likely in this implementation. The unreliability is an artifact of the Lites server which provides the microkernel with services that allow one to use UNIX applications on DTOS. The combination of the microkernel, Lites server, and the Security Server is prone to paging errors and deadlocks. To avoid these errors, the microkernel must have a sufficient set of permissions hard-coded into its cache (these permissions are not flushed from the microkernel). Some of the permissions required by the new Security Server to complete the hand-off must be in the hard-coded cache before the transition is initiated.

For example, the Security Server has page-able memory. During the hand-off, the Security Server may start using new areas of memory while processing a security request from the microkernel. If a page fault occurs, then the Security Server itself will request service from the microkernel. If the microkernel has not cached the permission required by the Security Server, it must in turn request a security computation from the Security Server. However, the Security Server is blocked on the request to the microkernel for service, and the microkernel cannot complete its request without the security computation from the Security Server. What makes these types of events unpredictable is the existence of other processes on the system that may request services from the Lites server while the security port rights are in transit. The new Security Server depends on the Lites server for services, but a thread of execution in the Lites server can be waiting for a security computation creating the deadlock.

Performance This is the slowest of the methods tested. During performance testing, a typical transition time (median) required 4.900 seconds, and all transitions fell with the range of 4.820 to 5.010 seconds. This might not be as fast as the Reload Policy method, but once again this would be more than satisfactory in systems such as the banking application mentioned in the introduction.

5.5 Adding Security Servers for New Tasks

The final method for changing the security policy is to create a set of task-based Security Servers. In the three previously described methods, all tasks operate under a single, monolithic policy. With this method there may be more than one Security Server computing access decisions for the microkernel and other clients, each defining a separate set of security rules. While the microkernel is enforcing multiple policies, each task on the system is associated with one and only one Security Server which serves as its primary Security Server. In fact, each task is associated with a list of Security Servers ordered by precedence. There is a well-defined policy for each task because there is only one way for each access request to be computed by the entire set of Security Servers.

For this method we introduce a new global variable: the Security Server stack. Each entry in the stack consists of a data structure containing the security and client ports for each Security Server. At boot time, the initial Security Server uses the **set_special_port** command to enter the security and client ports to the initial entry in the stack. Another global variable, **curr_ss**, points to that entry in the stack to indicate that the initial Security Server is the *current* Security Server. When another Security Server is created, it also enters its ports to the stack at the first available entry, and **curr_ss** is incremented to the next position in the stack.

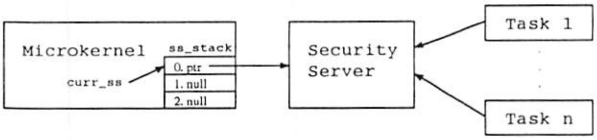


Figure 3: Security Server Stack Before "Push"

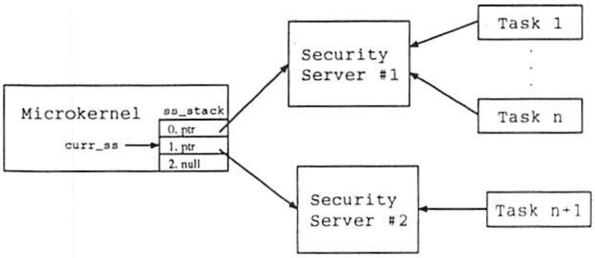


Figure 4: Security Server Stack After "Push"

Each task has a pointer labeled **ss_ptr** that identifies the Security Server that defines the policy under which the task is running. When tasks are created, **ss_ptr** is set to **curr_ss** by default, though the parent task may cause the value of **ss_ptr** for the new task to be set to the parent's Security Server. Like any other process, each new Security Server itself operates under the policy defined by a Security Server which precedes it in the stack (the Security Server immediately preceding it would be the default). If each Security Server in the stack refers to its immediate predecessor in the stack, then it is truly a stack-like implementation. If the Security Servers in the "stack" refer to servers older than their immediate predecessors, then a graph of the dependencies could be more accurately described as a "tree." This tree-like structure for the dependencies between the Security Servers give this implementation an interesting set of properties.

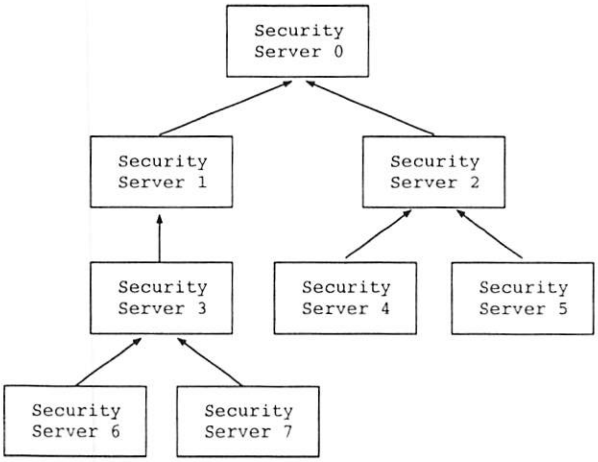


Figure 5: The tree-like dependency among a set of Security Servers

When the microkernel receives a request, it checks its cache for the permission. If the permission is not in the cache, it sends a request to the Security Server assigned to the requesting task. The Security Server computes the requested security access, unless it receives a request with a context that it does not understand. If the Security Server cannot resolve the SIDs into security contexts, it forwards the request to its own Security Server. The request is passed down the tree until some Security Server, possibly the "root" Security Server, is able to resolve the SIDs into contexts and a security computation can be made.

This method for changing the security policy is the most robust and possibly the most flexible method of the four methods discussed in this paper. However, the additional flexibility and reliability of enforcing multiple security policies may come with an increased cost for assuring the security of the system.

Policy Flexibility This method for changing policy provides the capability for considerable flexibility for changing the policy. However, as new Security Server s are created, only new tasks operate under the new policy rules; so changes to the system-wide policy are local rather than global. In other words: you can't teach an old dog new tricks, because old tasks will continue to run under the policy defined by the old Security Server.

There is the possibility that the stack could be augmented by using one of the other policy changing mechanisms to force old tasks to run under a new policy. For example, if there are two servers in the stack at positions 0 and 1, the Security Server at position 0 could hand off to a third Security Server which is identical to the Security Server in position 1. Thus, both servers operating would define the same policy, and the microkernel would be enforcing only one policy rather than two. [In fact, the first two servers could then exit, all tasks with pointers to the second Security Server would be re-directed to the server at position 0 (the third server), and the system would only have one Security Server as well as one policy.]

Functional Flexibility Functional flexibility is the greatest strength of the Security Server stack. Allowing running processes to run under their original policy is a way of "grandfathering" in their allowed accesses. Thus, in our banking example, if some user is actively working on a process at 5:00 PM which must be completed, but the bank's security policy is set to change to a more restrictive policy at that time, the user would be allowed to continue his task because the task is operating under the less restrictive policy. However, any attempt by a user to create new tasks after 5:00 pm would be subject to the new, more restrictive policy.

Security This method is a double-edged sword. It is possible that certain tasks which need to be highly constrained could operate under more restrictive policies than is generally required. This could be an advantageous design for increasing security. However, coordinating the necessary elements could be a nightmare for system designers and for any attempts to provide formal assurance evidence. In effect there would be multiple, overlapping security policies. One could not make broad global statements about the behavior of the system and the rules in place at any given time.

Also, once a task is granted a permission to perform some operation, it is allowed to keep that permission, even if another more restrictive Security Server is pushed onto the stack. Thus, in the event of an intrusion, a rogue process which has gained unauthorized access to system resources may be able to continue unchecked. Thus, the gains made for functional flexibility allow for a loss of security. In order to harden the defenses of a system like this, it would be necessary to graft another method of policy change on top of this one.

Reliability This method improves upon the hand-off for reliability because there is no vulnerable moment when the rights for the security port are in transit. It is also more reliable than the Reload Policy Method because the top Security Server in the stack will still be able to make security computations even if new Security Servers fail to initialize due to corrupted security databases.

Performance Explicit performance numbers are not available for this method. However, it is anticipated to be as fast or faster than the hand-off, and expected transition times should be between four and five seconds. The greatest factor in the performance for the stack is the loading of the large executable for creating a new server to push onto the stack, which is also true of the hand-off. The hand-off is slower because the rights to the security port have to be transferred from one server to the other. The Security Server stack is quicker than loading the executable for the new server, but adds an extra wait.

6 Conclusions

There are a number of ways of implementing adaptive security policies for security architectures which separate the definition of the policy from its enforcement. From the entire range of such implementations, this paper has examined four possible methods all of which have been implemented for the DTOS prototype by Secure Computing. Each implementation has strengths and weaknesses, and the trade-offs are encapsulated in Table 1 below. From the table, the server stack and the extended state appear to be the most attractive options for implementing adaptive security, but which choices one makes depends on the eventual application for the implementation as suggested below.

	Implementations			
Criteria	Reload Policy	Extended State	Hand-Off	Server Stack
Policy Flexibility	Fair	Good	Fair	Excellent
Functional Flexibility	Poor	Good	Fair	Excellent
Security	Good	Excellent	Fair	Poor
Reliability	Fair	Excellent	Poor	Good
Performance	Good	Excellent	Poor	Fair

Table 1: Summary of Trade-Offs

When applied appropriately, reloading the policy and the expanded state methods are the lightest weight implementations and provide good features for a narrow set of applications. In particular, the key features of these two methods are that they allow the Security Server to change the database without changing the algorithms from which the Security Server makes its security computations. The database and Security Server implementations for the expanded state have the potential to become complex. The additional complexity posed by this work may make alternate methods for implementation more attractive. The expanded state method is best left to small, incremental changes in the policy. By comparison, reloading the policy is probably not a an attractive option for system in which there are a large number of small changes to the databases since each change of policy would require its own database, and the issue of scalability may be burdensome.

The other two methods, the hand-off and server stack, allow for changes to the algorithms for computing permissions, and this is what accounts for the greater degree of policy flexibility. Because of the multiple points of control, the security server stack offers the greatest functional and policy flexibility, and the inheritance structure of the parent-child relationships between Security Servers offers the ability to grandfather permissions for running applications. However, that very same asset is a liability. Policy changes using the stack are local, not global. Thus, it is not possible to revoke permissions using that method alone. Furthermore, depending on the number of policies supported by the system, the security server stack holds the potential for being the heaviest weight implementation.

Not addressed in Table 1 is the possibility of mixing and matching the four methods described in this paper to capture the best security features of one method with the best flexibility features of another. For example, one might combine the security server stack and with the hand-off method in the following way. Tasks would operate under task-based policies with the server stack up to a certain point in time, allowing for local changes to the policy based on roles and tasks, and then a server might hand off to its parent and shut down. For example, in the banking application in which the more restrictive nighttime policy is the child of the less restrictive daytime policy (i.e., the stricter Security Server is pushed onto the stack at 5 PM), the nighttime server could hand off to its parent the following morning at 8 AM and shut down. Similarly, one might follow the hand-off or server stack by reloading the policy to change the internal tables of a Security Server without changing the fundamental algorithms by which it operates.

Current work on adaptive security has focused on theoretical aspects of adaptive security policies and on various mechanisms for implementing adaptive security. Future work on adaptive security policies should turn from the theoretical to the applied, hopefully by implementing a demonstration system. For example, one might implement a set of banking applications that would operate under policies for daytime and after-hours processing. A demonstration system of this type should also be accompanied by formal assurance evidence such as a formal security policy. However, until there is a real system to examine, formal assurance for adaptive security can only be speculative.

7 Acknowledgments

The authors would like to thank Rome Laboratory for sponsoring the contract under which this research was completed. Thanks also go to Todd Fine, Terry Mitchem, Duane Olawsky, and Ray Spencer for technical assistance, to Dan Thomsen and Joe Andert for comments on the manuscript, and to Cornelia Murphy who served as program manager.

References

[BK85] W.E. Boebert and R.Y. Kain. A Practical Alternative to Hierarchical Integrity Policies. In *Proceedings of the 8th National Computer Security Conference*, September 1985.

[BL73] D. Elliott Bell and Leonard J. La Padula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corporation, May 1973.

[BN89] David F. C. Brewer and Michael J. Nash. The Chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, CA, May 1989.

[CW87] David D. Clark and David R. Wilson. A comparison of commercial and military computer security policies. In *IEEE Symposium on Security and Privacy*, pages 184–194, Oakland, CA, April 1987.

[FCK95] David F. Ferraiolo, Janet A. Cugini, and D. Richard Kuhn. Role-based access control (rbac): Features and motivations. In *Computer Security Applications Conference*, New Orleans, LA, dec 1995.

[FM93] Todd Fine and Spencer E. Minear. Assuring Distributed Trusted Mach. In *Proceedings IEEE Computer Society Symposium on Research in Security and Privacy*, pages 206–218, May 1993.

[GHS97] Paula Greve, John Hoffman, and Richard Smith. Using Type Enforcement to Assure a Configurable Guard. In *Proceedings of the 13th Annual Computer Security Applications Conference*, 1997. To appear.

[Hof97] John Hoffman. Implementing RBAC on a Type Enforced System. In *Proceedings of the 13th Annual Computer Security Applications Conference*, 1997. To appear.

[Loe93] Keith Loepere. *OSF Mach Kernel Principles*. Open Software Foundation and Carnegie Mellon University, May 1993.

[Min95] Spencer E. Minear. Providing policy control over object operations in a Mach based system. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, pages 141–156, June 1995.

[Pfl97] Charles P. Pfleeger. *Security in Computing*. Prentice Hall, Inc., Upper Saddle River, NJ, 2 edition, 1997.

[Ras91] Richard F. Rashid. Mach: A case study in technology transfer. In Richard F. Rashid, editor, *CMU Computer Science: A 25th Anniversary Commemorative*, chapter 17, pages 411–421. ACM Press, 1991.

[SC96] Ravi S. Sandhu and Edward J. Coyne. Role-based access control models. *IEEE Computer*, pages 38–47, February 1996.

[Sec97] Secure Computing Corporation. DTOS Generalized Security Policy Specification. DTOS CDRL A019, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, June 1997.

[SKTC96] Edward A. Schneider, William Kalsow, Lynn TeWinkel, and Michael Carney. Experimentation with adaptive security policies. Technical report, Secure Computing Corporation, 2675 Long Lake Road, Roseville, Minnesota 55113-2536, January 1996. Final Report for Rome Laboratory contract F30602-95-C-0047.

The CRISIS Wide Area Security Architecture*

Eshwar Belani† Amin Vahdat† Thomas Anderson‡ Michael Dahlin§

Abstract

This paper presents the design and implementation of a new authentication and access control system, called CRISIS. A goal of CRISIS is to explore the systematic application of a number of design principles to building highly secure systems, including: redundancy to eliminate single points of attack, caching to improve performance and availability over slow and unreliable wide area networks, fine-grained capabilities and roles to enable lightweight control of privilege, and complete local logging of all evidence used to make each access control decision. Measurements of a prototype CRISIS-enabled wide area file system show that in the common case CRISIS adds only marginal overhead relative to unprotected wide area accesses.

1 Introduction

One of the promises of the Internet is to enable a new class of distributed applications that benefit from a seamless interface to global data and computational resources. A major obstacle to enabling such applications is the lack of a general, coherent, scalable, wide area security architecture. In this paper, we describe the architecture and implementation status of CRISIS, a wide area authentication and access control system. CRISIS forms the security subsystem of WebOS, a system that extends operating system abstractions such as security, remote process execution, resource management, and named persistent storage, to support wide area distributed applications.

Today, many wide area applications are limited by the lack of a general wide area security system. As one example, one of the goals of the WebOS project is to build a scalable SchoolNet service, to provide a safe place for the tens of millions of K-12 students in our states (California, Washington, and Texas) to learn and play. One challenge to making SchoolNet a reality is building scalable network services, for example, to provide a highly available email account to every student, without requiring a system administrator at every location. This paper focuses on an equally difficult challenge – maintaining the confidentiality and integrity of data and resources, for example, so that unauthorized people cannot obtain information about school children. As another example, we (as a geographically distributed development team) would like to use WebOS to allow us to seamlessly access any file or computational resource at any of our sites; once CRISIS is fully operational, we plan to rely on it to protect our development environment from external attacks. As a final example, we have built Rent-A-Server [Vahdat et al. 1997], a system to dynamically replicate and migrate Internet services, to gracefully handle bursty request patterns, and to exploit geographic locality to reduce latency and congestion. To be practical, however, RentAServer requires the ability to securely access and control remote data and computational resources (e.g., CPUs and disks).

An initial approach for supporting secure access to remote resources is to simply employ an authenticated login protocol. Unfortunately, this approach is inadequate because many wide area applications require more fine-grained control over access to remote resources. Further, the administrative overhead of creating and maintaining separate accounts in all domains where users wish to run jobs can be prohibitive. For example, it would be difficult to use authenticated login to support a user job running on an anonymous compute server in a remote administrative domain that needs access to a single file on the user's home file system.

Another approach for supporting secure wide area applications is to add fine-grained rights transfer to an existing authentication system, such as Kerberos [Steiner et al. 1988]. However, while Kerberos has proven quite successful for local area networks in a single adminis-

*This work was supported in part by the Defense Advanced Research Projects Agency (N00600-93-C-2481, F30602-95-C-0014), the National Science Foundation (CDA 9401156), Sun Microsystems, California MICRO, Novell, Hewlett Packard, Intel, Microsoft, and Mitsubishi. Anderson was supported by a National Science Foundation Presidential Faculty Fellowship. For more information, please see `http://now.cs.berkeley.edu/WebOS`, or send email to `vahdat@cs.berkeley.edu`.

†Computer Science Division, University of California, Berkeley

‡Department of Computer Science and Engineering, University of Washington, Seattle

§Computer Science Department, University of Texas, Austin

trative domain, it faces a number of challenges when extended to the wide area. First, Kerberos has no redundancy; security is undermined if even a single authentication server or ticket granting server is compromised, allowing an adversary to impersonate any principal that shares a secret with the compromised authentication server. In the wide area, the number of such single points of failure scales with the size of the Internet. Further, Kerberos requires synchronous communication with the ticket granting server in order to set up communication between a client and server; in the wide area, synchronous communication with a hierarchy of ticket granting servers is required. Given that the Internet today is both slow and unreliable, this can have a significant effect on availability and performance as perceived by the end-user. Although Kerberos servers could conceivably be replicated to improve availability, the servers would need to be geographically distributed to hide Internet partitions, providing an intruder even more points of attack.

Public key cryptography seems to hold out the promise of improving availability and security in the wide area, by eliminating the need for synchronous communication with a trusted third party. The public key of every principal (user or machine) can be freely distributed; provided the public keys are known, two principals can always communicate if they are connected, regardless of the state of the rest of the Internet. Unfortunately, this also comes at a cost; any compromise of a private key requires that every entity on the Internet be informed of the compromise. This is analogous to Kerberos, in that the number of single points of failure (in this case, the number of private keys) scales with the size of the Internet.

In this paper, we present the design and implementation of CRISIS, a system for secure, authenticated access to wide area resources. To avoid an ad hoc design where features are thrown together in an attempt to prevent all known types of security attacks, our approach is the systematic application of a set of design principles. These principles are inspired by analogy with other areas of distributed systems, where scalability, performance and availability can be achieved through redundancy, caching, lightweight flexibility, and localized operations. A goal of the CRISIS architecture is to demonstrate that these principles can also be applied to increasing the security of wide area distributed systems.

Specifically, the principles underlying the design of CRISIS include:

- Redundancy: There should be no single point of failure in the security system; any attack must compromise two different systems in two different ways. For example, every certificate (such as that identifying a user's public key) is revocable within a few minutes; thus, an attacker must not only steal a private key, but must do so without detection or must also corrupt the revocation authority. The notion of using redundancy to improve security is an old one, but it has not been systematically applied. For example, Internet firewalls are used to protect organizations from security attacks from the outside world, to hide the fact that most local operating systems are notoriously insecure. Unfortunately, this has reduced the pressure on local operating systems to improve their security, so that once inside a firewall, an attacker usually has nearly free reign. Similarly, Internet browsers purport to be able to safely execute Java applets, supposedly rendering traditional operating system security irrelevant. The ongoing discovery of security holes in Java verifier implementations [Dean et al. 1996, Sirer et al. 1997], however, has led us to run remotely executing programs in a restricted environment [Goldberg et al. 1996], *in addition to* using a verifier.

- Caching: CRISIS caches certificates to improve both performance and availability in the presence of wide area communication failures; while all certificates are revocable, they are given a revalidation timeout to hide short-term unavailability of the revocation authority due to reboots or Internet partitions. This principle was inspired by research in mobile file systems which argued that caching can improve availability, e.g., for disconnected operation [Kistler & Satyanarayanan 1992].

- Least Privilege: Users should have fine-grained control over the rights they delegate to programs running on their behalf, whether local or remote; to be useful, this control must be easy to specify. CRISIS provides two mechanisms to support least privilege: *transfer certificates*, limited unforgeable capabilities which allow for confinement and can be immediately revoked, and lightweight roles that can created by users without the intervention of a system administrator and without requiring changes to access control lists.

- Complete accountability: CRISIS logs all evidence used to determine if a request satisfies access control restrictions, locally at the machine guarding access. Most existing systems log only coarse-grained authentication information, making accountability in the face of rights transfer difficult (e.g., some systems may want to grant a request only if every member along a chain of delegation is trusted). Our design differs from previous efforts to add capability-like certificates to Kerberos [Neuman 1993], which require distributed logging by all ticket granting servers involved in propagating a request in the wide area.

- Local Autonomy: Each user identity is associated with a single administrative domain, and that domain is solely responsible for determining how and when user privileges are distributed beyond the local administrative domain. Each local domain is also responsible for determining the level of trust placed in any given remote domain. A design relying on deference to a global, centralized authority is not only less flexible, but less likely to be widely adopted [Birrell et al. 1986].
- Simplicity: Simple designs are easier to understand and implement. In the context of security, simplicity is especially important to minimize the probability of an security hole resulting from an implementation error. A provably secure but highly complex system architecture is unlikely to be either properly implemented by system designers or properly understood by end users (for example, leading to errors in setting up access control lists).

CRISIS is loosely based on the DEC SRC security model [Lampson et al. 1991]. Relative to their work, one of our contributions is to simplify the model by using *transfer certificates* as the basis of fine-grained rights transfer across the wide area. Transfer certificates provide an intuitive model for both rights transfer and accountability, as they allow a complete description of the chain of reasoning associated with a transfer of rights. In addition, *revocation* is a first class CRISIS operation; even privileges described by transfer certificates (which are typically valid only for a limited period of time) can be revoked immediately. CRISIS also provides for explicit reasoning about the state of loosely synchronized clocks, an important consideration for wide area applications. Further, CRISIS supports user-defined lightweight roles, to capture persistent collections of transferred rights (e.g., "Tom running a job on remote supercomputer"). Finally, in contrast to the DEC SRC work which was implemented in the kernel of a platform that is no longer available, CRISIS is designed to run portably across multiple platforms, a requirement for a wide area security system to be useful in practice.

The rest of this paper describes CRISIS in more detail. We first provide some motivating examples for CRISIS along with a quick review of relevant technology in Sections 2 and 3. We then outline the CRISIS architecture in Section 4, followed by a detailed example of how CRISIS is used in Section 5. We evaluate the performance of our implementation in Section 6, and discuss related work in Section 7. We summarize our results in Section 8.

2 Motivation

CRISIS is the security subsystem for WebOS [Vahdat et al. 1997]. The goal of WebOS is to provide operating system primitives for wide area applications now available only for a single machine or on a local area network. Such abstractions include authentication, authorization, a global file system, naming, resource allocation, and an architecture for remote process execution. To date, wide area network applications have been forced to re-implement these services on a case by case basis. WebOS aims to ease and support network application development by providing a substrate of common OS services.

The focus of this paper is the architecture of the WebOS security subsystem which cuts across all other aspects of the system. Below, we briefly describe a number of scenarios we have used to drive the CRISIS design:

- SchoolNet: One motivating example is to provide Internet services such as email, Web page hosting, and chat rooms for very large numbers of school children. One desirable feature of such a system is to allow geographically distributed children to be able to interact with one another, while keeping both the interactions and the identities of those involved private. Further, to be useful to school children, the security system must work with only limited direction from end users (e.g., you cannot trust a fifth grader to correctly set up access control lists).
- Wide Area Collaboration: Users in separate administrative domains should be able to collaborate on a common project. For example, a project's source code repository should be globally accessible to authorized principals for check in/check out; in addition, unique hardware (such as supercomputers) should be seamlessly accessible independent of geographic location.

- Geographically Distributed Internet Services: If it were easy to geographically replicate and migrate Internet services, end-users would see better availability, reduced network congestion, and better performance. Today, only the most popular sites can afford to be geographically distributed; for example, Alta Vista [Dig 1995] has mirror sites on every major continent, but these mirrors are physically administered by DEC, manually kept up to date, and visible to the end user. One of our goals is to make all this transparent, to make it feasible for third party system administrators to offer computational resources strategically located on the Internet for rent to content providers; in the limit, content providers could become completely virtual, with the degree and location of replicas dynamically scaled based on access patterns.

- Mobile Login: Users should be able to login and to access resources from any machine that they trust. Secure login requires mutual authentication. Thus, users will only log into machines certified to have been booted properly by a trusted system administrator. Likewise, local system administrators enforce which users are allowed login access (e.g. login to Berkeley by Stanford users would be disallowed outright). Finally, users should be allowed to adopt restricted roles representing the amount of trust they have for the machine being logged into.

- Encrypted Intermediate Caches: To improve application performance, untrusted third party servers may be utilized to cache encrypted private data. A special key would be created to encrypt the data rather than using the key of a particular principal; this key would then only be distributed to authorized users. One path to implementing such an application would be the use of Active Networks [Tennenhouse & Wetherall 1996] where intelligent routers can be utilized to perform the caching.

- Large Scale Remote Execution: Principals should be able to exploit global resources to run large scale computations. For example, NASA is placing petabytes of satellite image data on-line for use by earth scientists in predicting global warming. It is impractical to access this information using the current Internet "pull" model; scientists need to be able to run filters remotely at the data storage site to determine which data is useful for download. These filters should have access to necessary input (e.g., the filter executables) and output files (e.g., files into which the results are to be stored) on the scientist's machine, but to no other potentially sensitive data. Further, the remote computation environment should be protected from any bugs in the filters written by the scientists.

3 Background

One of the first steps in providing a secure Internet information system is to allow for encrypted, authenticated communication between arbitrary endpoints over an inherently insecure wide area network. Traditionally, the two choices for encryption and authentication are using secret key or public key cryptography. Encryption ensures an eavesdropping third party cannot alter the integrity or determine the content of the communication. Authentication allows for the identity of the principal at the opposite end of a communication link to be securely identified.

We choose public key over secret key (though one can be simulated with the other [Lampson et al. 1991]) because of the synchronous communication usually required by secret key systems. Secret key systems require a trusted third party that shares a secret with every potential communication endpoint. Although this requirement impacts system performance and availability by imposing an extra step in initiating communication, it is reasonable in the local area because the number of communication endpoints are limited and the network is more reliable. In the wide area, such a requirement strains system scalability because synchronous communication with a hierarchy of trusted third parties is required. Public key systems also require trusted third parties to produce certificates identifying principals with their public keys. However, these certificates can be cached (with a timeout), removing the need for synchronous communication with a third party to set up a communication channel. Allowing for direct communication in this fashion offers two advantages. First system availability is improved because an unavailable third party does not necessarily prevent communication. Second, system performance is improved by removing a communication step to a third party behind a potentially slow link.

In addition to public key encryption, we employ a number of other technologies to assist in development and to reduce the chance of introducing security flaws. We use Janus [Goldberg et al. 1996] to "sandbox" locally running applications that are not fully trusted. Janus runs at user-level, employing the UNIX System V `proc` file system to intercept potentially dangerous system calls and to disallow accesses outside of each process's defined sandbox. The implementation has negligible performance overhead and can sandbox unmodified applications. CRISIS also employs the SSL [Hickman & Elgamal 1995] protocol to provide transport network layer privacy and integrity of data, using encryption and message authentication codes. SSL supports a wide variety of cryptographic algorithms and is being deployed into wide area applications. Finally, as will be described in the next section, we use the X.509 syntax [Con 1989] to encode all certificates in CRISIS. The ITU-T Recommendation X.509 specifies the authentication service for X.500 directories, as well as the X.509 certificate syntax. The X.509 certificate syntax is supported by a number of protocols including PEM, S-HTTP, and SSL.

4 System Architecture

The goals of the CRISIS architecture can be described in two parts. First, users should be allowed secure access to global resources such as files, CPU cycles, or storage from anywhere in the world. Next, resource providers need mechanisms for authenticating those requesting their services and for authorizing those with the proper credentials. In this section, we provide a high-level view of the system architecture before detailing example usage in the next section.

In the following discussion, we assume the presence of three basic entities, adapted from the SRC logic [Lampson et al. 1991]:

- *Principals*: Principals are sources for requests. Examples of principals include users and machines. Principals make statements (requests, assertions etc.), have names, and can be associated with privileges.
- *Objects*: Objects are global system resources such as files, processors, printers, memory, etc.
- *Reference Monitors*: Once an access request from a principal to an object is authenticated, the reference monitor determines whether or not to grant the principal access to the object.

Consider the scenario where a user in California wishes to run a job at Texas which requires access to two input files. In turn, the job at Texas decides to subcontract a portion of its work to a machine in Washington. This sub-contracted work only needs access to the second input file. More formally, P_1 is a user in California, while P_2 and P_3 are machines willing to run jobs located at Texas and Washington respectively. O_1 and O_2 are objects (e.g. input files) located in California, with RM_1 and RM_2 their associated reference monitors. Assuming that P_1 (and only P_1) possesses access privileges to O_1 and O_2, consider the following sequence of events (summarized in Figure 1):

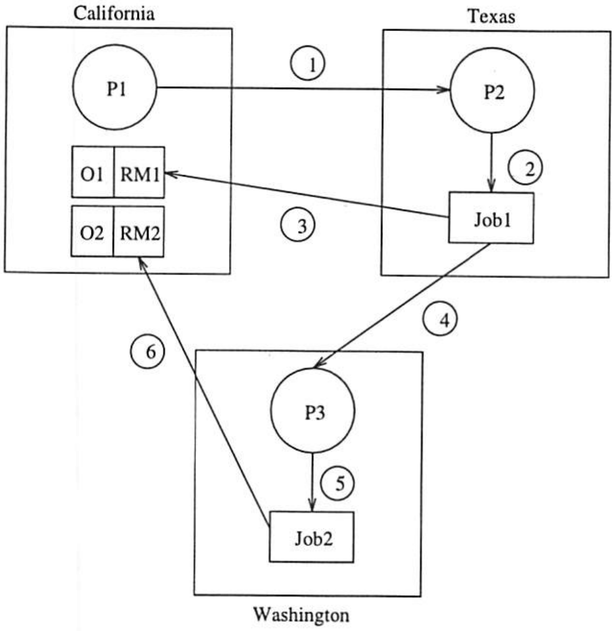


Figure 1: This figure describes a sample scenario where a user, P_1 requests a machine P_2 to run a job on its behalf. In turn P_2 sub-contracts a portion of the job to another machine P_3 in a separate administrative domain.

```
P1 states that P2 can access O1
and O2 until time T1.
P1 requests that P2 execute a
job on its behalf (steps 1 and
2).
P2 requests access to O1 from
RM1 (step 3).
P2 states that P3 can access O2
until time T2.
P2 requests that P3 execute a
job on its behalf (steps 4 and
5).
P3 requests access to O2 from
RM2 (step 6).
```

To carry out the above scenario in WebOS, the security system must support:

- Statements: Statements may be requests, declarations of privileges, or transfer of privileges. The identity of the principal making a statement must be verified, and all statements must be revocable.
- Associating privileges with processes: While a particular machine may possess a large set of privileges, individual processes only have access to a subset of these privileges. Similarly, users may only wish to grant a subset of their available privileges to each of their programs.
- Distributing statements across the wide area: A protocol for trust between different administrative domains must be established to allow for validation of privileges and identities across administrative boundaries.
- Time: The transfer of privileges can only be valid for a limited period of time. CRISIS requires a clear protocol for reasoning about time, and a methodology for isolating failures in cases where clock skews lead to a security breach.
- Authorization: When a reference monitor receives a request to access an object, it must determine the identity of the requester, ascertain the principal's privileges, and finally decide whether the request is authorized.

Our solutions to each of the above are described in the following subsections.

4.1 Validating and Revoking Statements

All statements in CRISIS, including statements of identity, statements of privilege, and transfer of privilege, are encoded in *certificates*. CRISIS certificates are signed by the principal making the statement and then countersigned by a principal of the signer's choosing. Each signature uses a separate timeout: the principal's signature is issued with a long timeout, while the countersignature is issued with a short timeout. The countersigner (i) checks if the statement has been revoked and (ii) refreshes its *endorsement* (by applying a new counter-signature with a new timeout to an expired certificate) of certificates, indicating that the rights are are still valid. Since we are building a public key system, the certificate's author need not be aware of the certificate's destination when it is created. Any principal with access to the certificate can determine the statement's author. Our certificates use the X.509 [Con 1989] standard format. CRISIS employs two basic types of certificates:

- *Identity Certificates*: An identity certificate associates a public key with a principal for a certain period of time. Depending on the type of principal (person, machine, process, etc.), an identity certificate can also specify a number of the principal's properties, such as name or organization.
- *Transfer Certificates*: Transfer certificates transfer a subset of a principal's privileges to another principal. A principal P_1 can use a transfer certificate to transfer to P_2 access rights to any objects it owns (e.g. O_1 and O_2). These transfers are expressed as a list of capabilities, resulting in arbitrary length certificates. Individual transfer certificates can be chained or they can disallow further transfers. Each successive link in a chain (e.g., from P_2 to P_3) can only refine, and never expand, the rights transferred. Transfer certificates are presented to reference monitors as proof of access rights. The reference monitor is able to verify the sequence of statements and the identity of each principal in the chain, ensuring complete accountability.

Of course, identity certificates must be signed by an authority trusted by both endpoints of a communication channel (see section 4.3 for the case where no single authority is so trusted by both parties). This trusted third party, called the Certification Authority (CA), maps public keys to principals and maintains a Certificate Revocation List enumerating all public keys that have changed or that have been knowingly compromised. In CRISIS, CA's sign all identity certificates with a long timeout (usually weeks) and identify a locally trusted on-line agent (OLA) responsible for counter-signing the identity certificate with a relatively short timeout (usually hours).

The redundancy of a split CA/OLA approach offers a number of advantages. First, to successfully steal keys, either both the OLA and CA must be subverted or the CA must be subverted undetected. By making key revocation a simple operation (as described below), we are able to pro-actively revoke keys when a CA comes under attack. Further, the CA is usually left off-line since certificates are signed with long timeouts, increasing system security since an off-line entity is more difficult to attack. Another advantage of the split CA/OLA approach is that a malicious CA is unable to revoke a user's key, issue a new identity certificate, and masquerade as the user without colluding with the OLA [Crispo & Lomas 1996]. Also, while a malicious OLA can mount a denial of service attack, the CA is still able to reissue new certificates employing a different OLA. Finally, this approach improves system performance because certificates can be cached for the timeout of the counter-signature, removing the need for synchronous three-way communication in the common case.

We generalize the OLA to make revocation a first class operation in CRISIS. All certificates are revocable modulo a timeout. To revoke a particular privilege, the OLA which endorses the certificate must be informed that the certificate should no longer be endorsed. Once the timeout period for the endorsed certificate expires, the rights described by the certificate are effectively revoked because the OLA will refuse re-endorsement for that certificate. Revocation is used not only for exceptional events such as stolen keys. For example, the rights of a remote job are revoked upon its completion or when a user decides to kill the job. As another example, privileges associated with a login session are revoked on user logout.

The use of transfer certificates in CRISIS also simplifies both the implementation of and reasoning about delegation, which allows one principal to act on behalf of a second principal. Such delegation is useful in many contexts. For example, a database server will receive requests from many users, with individual operations executed in the context of the rights of a single user. It is important that a user's privileges are not amplified by employing the rights of the server. Such delegation is difficult to properly design. For example, early versions of UNIX `sendmail` were *setuid root* to allow the program to write to any user's mailspool. However, a bug allowed users to write any system file to a mail message addressed to themselves.

In CRISIS, users sign transfer certificates allowing servers to act on their behalf for accessing files, running jobs, etc. Servers provide these certificates to reference monitors when making requests on behalf of a user (as opposed to certificates describing their own rights), reducing the chance of the server being granted access on its own behalf when acting on a user's behalf. Relative to the SRC system [Lampson et al. 1991], where reference monitors use a pull model to search for proof that a principal should be granted access, CRISIS transfer certificates reduce complexity and hence the chance that an implementation error will lead to unauthorized accesses.

4.2 Processes and Roles

4.2.1 Security Domains

Given the abilities to authenticate principals, CRISIS also requires a mechanism for associating privileges with running processes. Each CRISIS node runs a security manager responsible for mediating access to all local resources and for mapping credentials to *security domains*. In CRISIS, all programs execute in the context of a security domain. For example, a login session creates a new security domain possessing the privileges of the principal who successfully requested login. As will be described in Section 5.1, a security domain, at minimum, is associated with a transfer certificate from a principal to the local node allowing the node to act on the principal's behalf for some subset of the principal's privileges.

Processes are able to access wide area resources through resource providers responsible for managing each remote resource, such as processor cycles or disk space. In conjunction with security managers, resource providers determine the access privileges of processes requesting resources. CRISIS nodes currently run the following resource providers, each with their own set of reference monitors:

- *Process Managers*- A Process Manager is responsible for executing jobs on requested nodes. The Process Manager identifies the security domain associated with a request, obtains the credentials associated with the domain from the security manager, and then attempts to satisfy the request.
- *WebFS* - A WebFS server implements a cache coherent global file system. Similar to the Process Manager, upon receiving a file access request, the WebFS server determines the security domain from the security manager. Using this information, the WebFS server determines whether the access should be granted or denied.
- *Certification Authorities* - As described above, CA's take requests for creating identity certificates. The CA maintains a reference monitor with the list of principals authorized to create, modify, or invalidate identity certificates.

The interaction between resource providers, security domains, and security managers are described through the CRISIS protocols for login, file access, and remote process execution in Section 5.

4.2.2 Roles

In the wide area, it is vital for principals to restrict the rights they cede to their jobs. For example, when log-

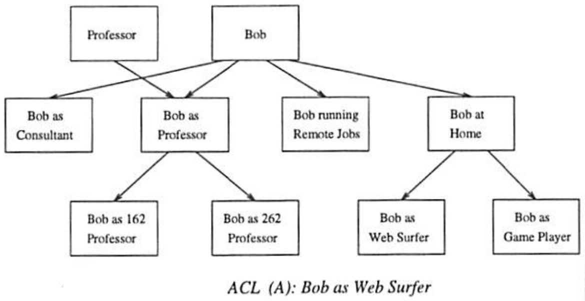


Figure 2: This figure describes how users may arrange their roles in a hierarchical fashion where each node in the tree possesses all the privileges of all of its direct descendants.

ging into a machine, a principal implicitly authorizes the machine and the local OS to speak for the principal for the duration of the login session. Whereas with private workstations, users generally have faith that the local operating system has not been compromised, confidence is more limited when using anonymous machines across the wide area, for example, to run large scale simulations. Roles associate a subset of a user's privileges with a name, allowing users a convenient mechanism for choosing the privileges transferred to particular jobs.

A principal (user) creates a new role by generating an identity certificate containing a new public/private key pair and a transfer certificate that describes a subset of the principal's rights that are transferred to that role; an OLA chosen by the principal is responsible for endorsing the certificates. Thus, in creating new roles, principals act as their own certification authority. The principal stores the role identity certificate and role transfer certificate in a *purse* of certificates that contains all roles associated with the principal. The purse is stored in the principal's home domain. While it is protected from unauthorized access by standard OS mechanisms, the contents of the purse are not highly sensitive since each entry in the purse simply contains a transfer certificate naming a role and potentially describing the rights associated with that role. The principal also stores each role's private key —encrypted by a password unique to the role —in the file system.

CRISIS roles are more lightweight than the roles described in other security systems (e.g., [Lampson et al. 1991]). First, they can be created by the user without requiring intervention from a centralized authority, allowing the CA to remain off-line. Next, while ACLs can be modified to describe a particular role's privileges, roles can also act as persistent lightweight capabilities. The transfer certificate used to create the role can describe the exact access rights possessed by the role (e.g., read access to files A, B, and C).

Further, transfer certificates can be used to arrange roles in a hierarchy, with the principal's most privileged role serving as the hierarchy's root. Such a hierarchy can be used in two ways. In a *single inheritance* model, each role possesses a strict subset of its parent's privileges. Thus, ACLs can be used to describe the "minimum" privilege level required to access a given object (a role is given access to an object only if it is a direct ancestor of a role in the object's ACL). With a *multiple inheritance* model, roles can draw upon the privileges of multiple ancestors. Figure 2 presents an example of such a hierarchy and applications of the two models. The object, *A*, can only be accessed by *Bob as Web Surfer* or one of its direct ancestors (*Bob at Home* or *Bob*). The Figure also illustrates that *Bob as Professor* (and its descendants) inherits privileges from both *Bob* and the generic, *Professor*. This may be useful, for example, to express that professors are able to read student accounts but to allow such access to Bob only when he is acting as a professor.

In CRISIS, creating a new group is similar to creating a new role. A principal creates a new group by acting as a CA to create an identity certificate naming the new group. The creating principal then signs transfer certificates to all group members, specifying both membership and any update privileges associated with the group, for example, whether the member has the ability to add or remove other group members. The newly created group name can then appear on ACLs like any other principal name.

4.3 Hierarchical Trust

We assume the presence of multiple, autonomous administrative domains, and that each domain has at least one trusted CA/OLA pair. CA's in different administrative domains are not equally trusted. Thus, CA's are arranged hierarchically, with individual CA's determining which parents, siblings, or children are trusted (and to what extent). The hierarchical arrangement of CA's builds on our model of implementing roles, where principals act as CA's in creating roles with the locally trusted CA acting as the principal's parent in a global hierarchy.

The manner in which the hierarchy is traversed is based on the theory presented in [Birrell et al. 1986]. In this model, a CA cannot speak for a principal who belongs to a descendant's domain, allowing separate administrative

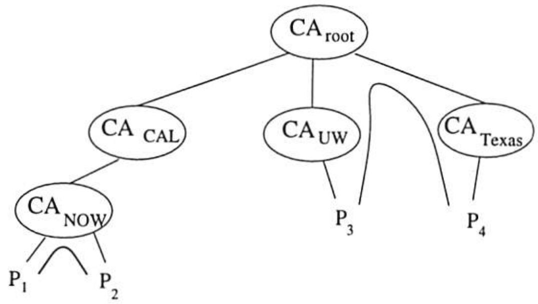


Figure 3: This figure describes how principals in different administrative domains can mutually authenticate. A path of trust is established through the principals' least common ancestor.

domains to maintain local autonomy. Thus, a principal receiving a certificate endorsed by a CA in a foreign administrative domain believes the certificate valid only if a *path of trust* is present from the local domain to the remote domain. The presence of such a path is determined by traversing the least common ancestor of the two domains in the CA hierarchy. Principals trust their local CA more than any of the CA's ancestors in the CA hierarchy. Thus, if an ancestor of a CA is compromised, transactions among local principals are not affected, increasing system availability and keeping trust as local as possible.

Figure 3 depicts an example of the arrangement of principals in multiple administrative domains. In this example, Principals $P3$ and $P4$ must establish a path of trust through the root CA to successfully authenticate one another. Demonstrating the principle of locality of trust, Principals $P1$ and $P2$ need only establish a path of trust through their common ancestor one level up to mutually authenticate.

4.4 Time

Since all CRISIS certificates contain timeouts and since these certificates are distributed across the wide area, the system must make assumptions about clock synchronization. Further, CRISIS must protect against security attacks exploiting a node's notion of time. If time on a machine is corrupted, statements can be used beyond their period of validity.

Today, most workstations possess fairly accurate clocks that are periodically synchronized with any of a number of external sources. However, time-sensitive applications (and hence, reference monitors) may require guarantees above and beyond such loose synchronization, for example, that the local clock is periodically synchronized with a trusted external source. Other applications will require an invoice of all assumptions made during a computation in the case where data is corrupted or leaked to determine the exact cause of the corruption, assuring complete accountability.

For CRISIS, we assume the presence of replicated, trusted time servers. Principals producing certificates with timeouts (e.g., CA's and OLA's) contact these servers periodically to obtain signed certificates containing the current time to validate the principal's notion of time. If the principal's time differs by more than a few seconds (i.e., within network delay bounds) from the time supplied by the server, the principal assumes that either the time server or the local operating system/hardware has been compromised (to determine which, a second server might be contacted). Such communication with time servers need not be synchronous, since the time certificates can be cached to prove recent synchronization.

In CRISIS, time certificates are provided to resource managers to prove that a node's notion of time closely matches the value reported by a trusted time server at some recent point in the past. CRISIS identity and transfer certificates report time values (such as expiration time) relative to the value contained in a chained time certificate. While use of time certificates does not guarantee that time-based attacks can be avoided or prevented, it can aid in determining the cause of certain security violations *post-mortem*. Thus, if a security breach is detected, analysis of certificates used to gain unauthorized access can be used to determine the cause of the attack. For example, examination of the certificates may show that a node attempted to use an expired time certificate or that a time server was compromised and reported faulty values of time.

4.5 Authorization

Once a request has been securely transmitted across the wide area, and properly authenticated, the remaining task is *authorization*, determining whether the principal making the request should be granted access. Traditionally, both Access Control Lists (ACLs) and capabilities have been used to describe the set of principals authorized to access a particular object. Since both ACLs and capabilities have advantages in different situations, we use a hybrid model similar to that proposed in [Neuman 1993].

The targets of CRISIS ACLs are service-specific. Currently, file ACLs contain lists of principal's authorized for read, write, or execute access to a particular file. Process execution ACLs are a simple list describing all principals permitted to run jobs on a given node. CA ACLs contains the list of principals authorized to update, modify, or revoke identity certificates.

A process requests access to an object by contacting the object's reference monitor. In CRISIS, reference monitors are implemented on a service-by-service (e.g., file service) basis and form separate modules in the security manager. For example, the WebFS reference monitor is a separate module in the CRISIS security manager.

CRISIS takes a push-based approach to providing credentials for authorization: requesters are responsible for proving they are authorized to access to an object. Thus, principals transmit to reference monitors their request in conjunction with a list of certificates describing their credentials. This list of certificates may simply contain the requester's identity certificate or may contain a more elaborate set of transfer certificates. The alternative to push, a pull-based mechanism where the reference monitor requests necessary credentials from principals, can provide more flexibility; however, it also complicates system design and can reduce performance.

To determine whether a request for a particular operation should be authorized, the reference monitor first verifies that all certificates are signed by a public key with a current endorsement from a trusted CA and OLA. In doing so, the reference monitor checks for a path of trust between its home domain and the domains of all signing principals (as described in Section 4.3). In the common case, the existence of such paths is cached. The reference monitor then checks that none of the timeouts have expired and that time is reported relative to a value stated by a trusted time server (again by checking for a path of trust to the time server).

Once the above steps are taken, the reference monitor is ensured that all certificates are well-formed and valid. Given this knowledge, the reference monitor then reduces all certificates to the identity of single principals. For transfer certificates, this is accomplished by working back through a chain of transfers to the original granting principal. The requesting principal is able to act on the behalf of the reduced list of principals. Finally, the reference monitor checks the reduced list of principals against the contents of the object's ACL, granting authorization if a match is found.

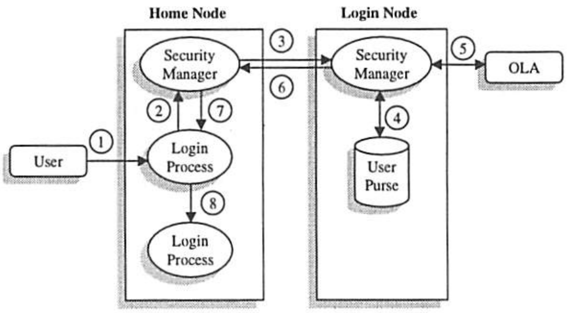


Figure 4: This figure details the steps used in CRISIS to authenticate a principal and authorize the principal for login.

5 CRISIS Protocols

Given the above high level description of the CRISIS architecture, we will now describe how the various system components interact to allow secure execution of routine tasks, including login, file access, and job execution (operations that potentially cross machine and/or administrative boundaries).

5.1 Login

The goal of login is to authenticate a principal to a node and to create a shell process with the principal's privileges. We achieve this by associating a security domain on the login node with a transfer certificate granting the node the privileges of the login role. We assume that each role is associated with a *home domain* and that users wishing to log in must authenticate their identity to their home domain. By minimizing the trust placed in the login node and by choosing a role with an appropriately small set of privileges, we enhance security and reduce the danger of key compromise (private keys never leave the home node). Further, the home domain possesses autonomy in determining the set of sites where principals are allowed to log in. The disadvantage of this approach is that all attempts to authenticate the user must involve the home domain, potentially decreasing system availability. In the future, we plan to investigate using smart cards in place of home domain machines to address this issue; we outline how this scheme would be integrated in the CRISIS architecture at the end of this subsection.

Initially, we consider the following login scenario: a principal accesses a shared workstation by entering a globally unique role name (e.g., *remote_tom@cs.washington.edu*). This role corresponds to the level of trust the principal places in the login node, and to the amount of rights required to successfully complete the desired tasks, for example, reading mail. Once the role is chosen, the principal trusts the OS of the login node with all the privileges associated with that role, since the OS is free to masquerade as the role (at least for the duration of the granted transfer certificate). The login sequence is described in Figure 4 and is summarized below:

1. The principal types in a suitable role name to the login process and enters the password for that role.
2. The login process sends the role name to the local security manager.
3. The security manager at the login node determines the home domain of the specified role (currently explicitly described in the role name) and contacts the security manager at the role's home domain. The two security managers mutually authenticate using SSL and a trusted hierarchy of CA's and online agents.

   As part of this mutual authentication, the login node transmits a certificate signed by a local system administrator stating that the administrator believed that the login node had not been tampered with at boot time. The home node uses this information to aid in the login authorization decision.
4. The home domain uses the password to decrypt the locally stored private key for the specified role name. If the key is successfully decrypted, the home security manager looks up the credentials associated with the specified role in the principal's certificate purse.
5. The certificates are presented to the home domain OLA for endorsement. The OLA sends back the endorsed certificates. The home domain's security manager can optionally update the principal's purse with the endorsements.
6. The home domain signs transfer certificates (on the principal's behalf), transferring all the privileges associated with the specified role to the security manager on the login node
7. The result of the login request is returned to the login process.
8. If the login is successful, the login process creates a login shell for the user. The security manager creates a new security domain, associating the login shell with the set of certificates transmitted by the home node.

For a successful login, the result of the above sequence of steps is to allow the login node to act on behalf of the role for a time period determined by the home domain's security manager. Any processes spawned by the login shell are by default assigned to the same security domain. The protocols employed to access resources through this security domain are detailed in the next two subsections.

One limitation of the above scheme is that the login node is trusted with the role's password (though not its private key). A well-behaved machine will erase the password from memory as soon as it is transmitted to the home domain. Similarly, the local file and memory cache should be flushed upon logout to ensure that private state is not leaked even if the machine is compromised at a later time. Another limitation with the protocol is that the principal's home domain must be available at the time of login (i.e. no network failures/partitions), or authentication becomes impossible. We believe that both of these limitations are inherent given the current state of hardware/software systems. However, our design also supports the use of specialized, trusted hardware (such as smart cards or a portable computer) to enhance security (keep password from local machine) or higher availability (no need to contact home domain for login) or both.

A trusted hardware proxy, such as a smart card or a portable computer, can also be used to separate the tasks of authentication (principals proving their identity) and authorization (determining that the principal is privileged to login to the remote machine with the specified role)[1]. The proxy can store both a role's private key and the associated password to implement a challenge/response protocol at login as follows[2]. When the home domain is notified of a login attempt, it encrypts a random number in the role's public key and transmits the result to the login machine's security manager. The proxy prompts the user for a password needed to de-

[1] Even if a smart card is used for authentication, it may still be desirable to require joint endorsement of a login session from both the target login machine and the user's home domain. Thus, if remote login is locally authorized, the home domain may disallow the login as a matter of policy. For example, login to a competitor's machine may be disallowed to prevent spoofing attacks.

[2] We present one simple scheme; other zero-knowledge algorithms such as Fiat-Shamir [Fiat & Shamir 1987] could also be utilized.

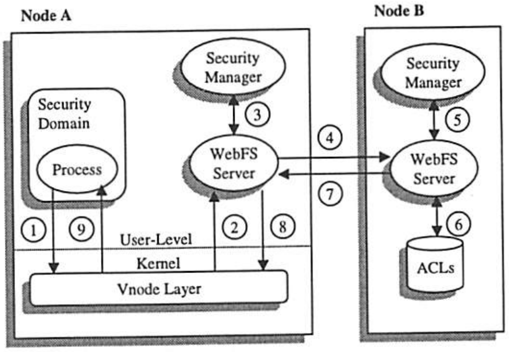


Figure 5: This figure describes the sequence of operations in accessing a remotely stored file through WebFS, a global file system.

crypt a locally stored (but encrypted) private key file for the role. The private key is used to decrypt the number, which is then transmitted back to the home domain. If the correct number is returned, the process of producing proper transfer certificates is followed as enumerated above. As a separate optimization, the task of authorization can be co-located with the hardware proxy to improve both the performance and availability of the login process (while trading off centralized autonomy in authorizing the locations from where particular roles are allowed login).

5.2 Accessing a Remote File

In this section, we demonstrate how the privileges associated with a CRISIS security domain are used to access a remotely stored file. While our techniques are general, we restrict our discussion to our specific implementation environment. We have built a global file system, WebFS [Vahdat et al. 1997] that allows read/write access to files stored across the wide area. WebFS is implemented at the vnode level [Kleiman 1986], similar to other distributed file systems such as NFS [Walsh et al. 1985] or AFS [Howard et al. 1988].

To illustrate the protocol for secure file access, we consider the scenario where a process running on Node A attempts access to a file located on Node B. The example is described in Figure 5, with the individuals steps detailed below:

1. A user process performs an `open` system call on a WebFS file stored on node B (currently WebFS employs a URL-like hierarchy for naming, e.g., /http/B/foo specifies a file foo stored on node B). The kernel translates this call into a `NodeAccess` operation in the Vnode layer.

2. The Vnode layer makes an upcall to a user-level WebFS server to carry out the access request (mode of WebFS functionality is implemented at user-level for ease of debugging and implementation).

3. The WebFS server contacts the local security manager with the requesting `uid/pid` pair to ascertain the privileges associated with the process attempting access to the remote file. For UNIX, The security manager maintains mappings between `uid/pid` pairs and security domains which in turn map to a set of transfer certificates describing the process's privileges.

4. The WebFS server on node A establishes an SSL connection with the WebFS server on node B, transmitting its own credentials and the credentials of the process requesting file access.

5. The WebFS server on node B contacts its local security manager to validate the transmitted certificates and to establish any necessary paths of trust to the potentially remote administrative domain containing node A.

6. Local file ACLs are consulted to determine if the requesting process possesses the request access privileges (e.g. read, write, or execute).

In steps 7-9, the result of the ACL check is returned through the WebFS server on node A, the vnode layer, and finally as the return value to the original `open` system call.

One concern with any system that allows file access from potentially untrusted machines is that local operating systems must be trusted with the contents of the file. That is, a corrupted operating system (or the local CRISIS security manager for that matter) could allow access to unauthorized users on the same host. Worse, if a machine is compromised after a user logs out, sensitive data could still be lost by inspecting the file/virtual memory cache. CRISIS employs two techniques to address these concerns. First, the CRISIS log out process discards the cache of any user accessed files through a WebFS system call. Next, for remote access to highly sensitive data, CRISIS allows the use of trusted portable computers running CRISIS software supporting mobile login. Using this technique, files are transmitted encrypted end-to-end until they reach the portable, at which point they can be decrypted and cached locally with a higher degree of security.

5.3 Running a Remote Job

Conceptually, the process of authenticating and authorizing execution of jobs on remote machines is similar to the process of remote file access. Currently, WebOS uses a UNIX command line program to request remote execution. This request results in a CRISIS library call, which contacts the local process manager with the identity of the principal. The protocol for process execution then proceeds similarly to the file access example described in the previous subsection.

Currently, the ACLs for remote process execution simply include the names of principals which have access to execute programs on a remote node. In the future, we plan to use transfer certificates and ACLs to contain information which specify the portion of the resources a certain role can consume. Another avenue for future work is building an interface to allow principals to reason about the set of privileges required by remote jobs. Clearly, remote jobs should run with the minimum set of privileges necessary to complete their task. However, determining this minimal set can be difficult. We plan to build an interface that allows users to run jobs locally to identify the minimal set of privileges that should be transferred to the job when it is run remotely.

Once a job execution request is authorized, CRISIS uses Janus [Goldberg et al. 1996] to set up a virtual machine to execute the process on the target machine, reducing the risk of violating system integrity. The Janus profile file describing the level of restriction imposed by the virtual machine is generated on the fly based on the identity of the requesting principal and the requirements of the job to be executed. Once set up, the virtual machine is associated with a CRISIS security domain, associating the virtual machine with the set of privileges specified by the principal requesting process execution. By both restricting jobs to originate from authorized users and placing running jobs in a sandbox, the local machine is protected from malicious or buggy programs even if the program's execution is requested from an authorized principal.

6 Performance

To quantify the performance impact introduced by CRISIS, we measured our global file system, WebFS, both with and without CRISIS enhancements. We measure the time required to read and write both 1 byte and 10 MB to a remote file. Measurements were taken between two Sun Ultrasparc 1's connected by a 10 Mb/s switched Ethernet.

Operation	*NFS*	*WebFS*	*WebFS w/CRISIS*
Read 1 byte	3 ms	47 ms	55 ms
Write 1 byte	100 ms	289 ms	340 ms
Read 10 MB	9.8 s	11.0 s	12.2 s
Write 10 MB	9.2 s	12.8 s	14.0 s

Table 1: This table describes the overhead introduced by adding CRISIS security to WebFS, a global file system. CRISIS file transfers are encrypted through SSL.

Table 1 summarizes our results. The first column describes performance for accessing uncached NFS files. The second column describes access to uncached files through a version of WebFS without CRISIS modifications. The added overhead of WebFS relative to NFS is caused by kernel to user-level crossings for cache misses (WebFS network communication code is implemented at the user-level for ease of implementation and debugging). The third column describes performance of WebFS with CRISIS security enhancements. We believe the 10-20% slowdown relative to the baseline WebFS to be acceptable given the added functionality of access control checks and encrypted file transfer.

The measurements in the third column reflect the case where user credentials are cached on the remote node. An additional 175 ms overhead is introduced to establish an SSL connection and 230 ms are required to transfer and cache an identity plus a single transfer certificate if user credentials are not cached remotely. Once again, this total 400 ms overhead is a one-time cost incurred the first time a user makes any access to a remote site (WebFS maintains a "cache" of active SSL connections between machines to avoid the cost of re-establishing an SSL connection for each access). Finally, read access to a cached 1 byte file through WebFS with CRISIS enhancements takes 720 μs, and reading a cached 10 MB file takes 170 ms, values comparable to cached access through NFS. In summary, our security enhancements introduce significant overhead for initial and uncached access because of switching to a user-level process for communication and the overhead of establishing an SSL connection for transmission of certificates. However, the common case read access to a cached file stays entirely in the kernel and provides performance comparable to a file system such as NFS.

7 Related Work

The conceptual framework of our security architecture is largely based on the theory presented in [Lampson et al. 1991]. In the introduction, we discussed the relationship between the DEC security work and our own. In this section, we describe a number of other efforts related to CRISIS.

SDSI [Rivest & Lampson 1996] is a distributed security infrastructure based on public keys with goals similar to our own. Their emphasis is on defining a standard format for certificates, rights transfer, and name spaces to provide a general security framework for Internet applications. With minimal extensions, SDSI could support CRISIS transfer certificates and remote execution of programs. Our work, however, is the largely orthogonal task of defining how such a framework can be used to provide redundant, high performance, and available security mechanisms for applications requiring secure remote control of wide area resources.

Neuman [Neuman 1993] discusses distributed mechanisms for authorization and accounting. Neuman's work has much the same vision as our own, namely limited capabilities in addition to ACL's. His work proposes a more general capability model. However, the capabilities are not auditable because proxies do not carry a chain of transfers. Further, Neuman's work is secret key as opposed to public key, meaning that synchronous communication is required for each transfer of rights. The trusted third party is responsible for recording transfers and transferring the end result. For example, if P_1 transfers rights to P_2, and P_2 further transfers rights to P_3, the trusted third party only passes on P_1 transferring rights to P_3 to any end reference monitors.

Jaeger and Prakash [Jaeger & Prakash 1995] present a model for discretionary access control in a wide area environment. In their work, principals specify the subset of their privileges that are to be transferred to a script written by a potentially untrusted third party. The actual rights transferred are negotiated between the application writer and the user. In their system implementation in Taos [Wobber et al. 1993] (a secure OS based on [Lampson et al. 1991]), they add dynamic principals for running programs with some subset of a principal's privileges, observing the difficulty of creating temporary principals and updating all necessary ACLs with the new principal name. Their dynamic principals are similar to one of the applications of CRISIS transfer certificates.

The goals of the Legion [Wulf et al. 1995] project are similar to our own in WebOS. In Legion, distributed computation takes place in the context of a distributed object system. Their approach to security is orthogonal to our own, with their primary goal being flexibility. Each legion object is able to implement its own security policy. Presumably, a number of base policies will be implemented which will suit the needs of a vast majority of applications. We believe that flexibility in the security system is a desirable feature; our approach in CRISIS can be viewed as one implementation of security for Legion objects.

8 Conclusions

In this paper, we have described the architecture of CRISIS, a security system for wide area applications. In designing CRISIS, we have endeavored to systematically apply principles from related fields to increase system security, availability, and performance across the wide area. These principles include: redundancy, caching, local autonomy, least privilege, and complete accountability. This paper describes how these principles have influenced our design and details the specific protocols used to carry out common operations across the wide area. Relative to earlier efforts, CRISIS uses transfer certificates as a simple mechanism for lightweight creation of roles and capabilities. While the current implementation runs only on Solaris, we expect to port the system to other platforms in the near future.

Acknowledgments

We would like to thank Neal Cardwell, Adam Costello, Terence Spies, and Chad Yoshikawa for participating in the many discussions which lead to our system architecture. We would also like to thank Brian Bershad, Neal Cardwell, Brian Dewey, Robert Grimm, and the anonymous referees for their valuable feedback and suggestions on earlier drafts of this paper.

References

[Birrell et al. 1986] A. Birrell, B. Lampson, R. Needham, and M.Schroeder. "Global authentication without global trust". In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, May 1986.

[Con 1989] Consultation Committee, International Telephone and Telegraph, International Telecommunications Union. *The Directory–Authentication Framework*, 1989. CCITT Recommendation X.509.

[Crispo & Lomas 1996] B. Crispo and M. Lomas. "A Certification Scheme for Electronic Commerce". In *Security Protocols International Workshop*, pp. 19–32, Cambridge UK, April 1996. Springer-Verlag LNCS series vol. 1189.

[Dean et al. 1996] D. Dean, E. Felten, and D. Wallach. "Java Security: From HotJava to Netscape and Beyond". In *Proceedings of the IEEE Symposium on Security and Privacy*, 1996.

[Dig 1995] Digital Equipment Corporation. *Alta Vista*, 1995. `http://www.altavista.digital.com/`.

[Fiat & Shamir 1987] A. Fiat and A. Shamir. "How to prove yourself: Practical solutions to identification and signature problems". In *Advances in Cryptology - Crypto '86*, pp. 186–194. Springer-Verlag, 1987.

[Goldberg et al. 1996] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. "A Secure Environment for Untrusted Helper Applications". In *Proceedings of the Sixth USENIX Security Symposium*, July 1996.

[Hickman & Elgamal 1995] K. Hickman and T. Elgamal. "The SSL Protocol". In *Internet RFC Draft*, 1995.

[Howard et al. 1988] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. "Scale and Performance in a Distributed File System". *ACM Transactions on Computer Systems*, 6(1):51–82, February 1988.

[Jaeger & Prakash 1995] T. Jaeger and A. Prakash. "Implementation of a Discretionary Access Control Model for Script-based Systems". In *Proc. of the 8th IEEE Computer Security Foundations Workshop*, pp. 70–84, June 1995.

[Kistler & Satyanarayanan 1992] J. J. Kistler and M. Satyanarayanan. "Disconnected Operation in the Coda File System". *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[Kleiman 1986] S. R. Kleiman. "Vnodes: An Architecture For Multiple File System Types in SUN UNIX". In *Proceedings of the 1986 USENIX Summer Technical Conference*, pp. 238–247, 1986.

[Lampson et al. 1991] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. "Authentication in Distributed Systems: Theory and Practice". In *The 13th ACM Symposium on Operating Systems Principles*, pp. 165–182, October 1991.

[Neuman 1993] B. C. Neuman. "Proxy-Based Authorization and Accounting for Distributed Systems". In *Proceedings of the 13th International Conference on Distributed Computing Systems*, May 1993.

[Rivest & Lampson 1996] R. L. Rivest and B. Lampson. "SDSI–A Simple Distributed Security Infrastructure". `http://theory.lcs.mit.edu/cis/sdsi.html`, 1996.

[Sirer et al. 1997] E. G. Sirer, S. McDirmid, B. Pandey, and B. N. Bershad. "Kimera: A Java System Architecture". `http://kimera.cs.washington.edu/`, 1997.

[Steiner et al. 1988] J. G. Steiner, B. C. Neuman, and J. I. Schiller. "Kerberos: an authentication service for open network systems". In *Usenix Conference Proceedings*, Dallas, Texas, February 1988.

[Tennenhouse & Wetherall 1996] D. Tennenhouse and D. Wetherall. "Towards an Active Network Architecture". In *ACM SIGCOMM Computer Communication Review*, pp. 5–18, April 1996.

[Vahdat et al. 1997] A. Vahdat, P. Eastham, C. Yoshikawa, E. Belani, T. Anderson, D. Culler, and M. Dahlin. "WebOS: Operating System Services For Wide Area Applications". UCB Technical Report CSD-97-938, December 1997.

[Walsh et al. 1985] D. Walsh, B. Lyon, G. Sager, J. M. Chang, D. Goldberg, S. Kleiman, T. Lyon, R. Sandberg, and P. Weiss. "Overview of the Sun Network File System". In *Proceedings of the 1985 USENIX Winter Conference*, pp. 117–124, January 1985.

[Wobber et al. 1993] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. "Authentication in the Taos Operating System". In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pp. 256–269, December 1993.

[Wulf et al. 1995] W. A. Wulf, C. Wang, and D. Kienzle. "A New Model of Security for Distributed Systems". University of Virginia CS Technical Report CS-95-34, August 1995.

Bro: A System for Detecting Network Intruders in Real-Time

Vern Paxson
Network Research Group
*Lawrence Berkeley National Laboratory**
Berkeley, CA 94720
vern@ee.lbl.gov

Abstract

We describe Bro, a stand-alone system for detecting network intruders in real-time by passively monitoring a network link over which the intruder's traffic transits. We give an overview of the system's design, which emphasizes high-speed (FDDI-rate) monitoring, real-time notification, clear separation between mechanism and policy, and extensibility. To achieve these ends, Bro is divided into an "event engine" that reduces a kernel-filtered network traffic stream into a series of higher-level events, and a "policy script interpreter" that interprets event handlers written in a specialized language used to express a site's security policy. Event handlers can update state information, synthesize new events, record information to disk, and generate real-time notifications via *syslog*. We also discuss a number of attacks that attempt to subvert passive monitoring systems and defenses against these, and give particulars of how Bro analyzes the four applications integrated into it so far: Finger, FTP, Portmapper and Telnet. The system is publicly available in source code form.

1 Introduction

With growing Internet connectivity comes growing opportunities for attackers to illicitly access computers over the network. The problem of detecting such attacks is termed *network intrusion detection*, a relatively new area of security research [MHL94]. We can divide these systems into two types, those that rely on audit information gathered by the hosts in the network they are trying to protect, and those that operate "stand-alone" by observing network traffic directly, and passively, using a packet filter. In this paper we focus on the problem of building stand-alone systems, which we will term "monitors." Though monitors necessarily face the difficulties of more limited information than systems with access to audit trails, monitors also gain the major benefit that they can be added to a network without requiring any changes to the hosts. For our purposes—monitoring a collection of several thousand heterogeneous, diversely-administered hosts—this advantage is immense.

Our monitoring system is called Bro (an Orwellian reminder that monitoring comes hand in hand with the potential for privacy violations). A number of commercial products exist that do what Bro does, generally with much more sophisticated interfaces and management software [In97, To97, Wh97],[1] and larger "attack signature" libraries. To our knowledge, however, there are no detailed accounts in the network security literature of how monitors can be built. Furthermore, monitors can be susceptible to a number of attacks aimed at subverting the monitoring; we believe the attacks we discuss here have not been previously described in the literature. Thus, the contribution of this paper is not at heart a novel idea (though we believed it novel when we undertook the project, in 1995), but rather a detailed overview of some experiences with building such a system.

Prior to developing Bro, we had significant operational experience with a simpler system based on off-line analysis of `tcpdump` trace files. Out of this experience we formulated a number of design goals and requirements:

High-speed, large volume monitoring For our environment, we view the greatest source of threats as external hosts connecting to our hosts over the Internet. Since the network we want to protect has

*This work was supported by the Director, Office of Energy Research, Office of Computational and Technology Research, Mathematical, Information, and Computational Sciences Division of the United States Department of Energy under Contract No. DE-AC03-76SF00098.

[1] Or at least appear, according to their product literature, to do the same things—we do not have direct experience with any of these products.

A somewhat different sort of product, the "Network Flight Recorder," is described in [RLSSLW97, Ne97].

a single link connecting it to the remainder of the Internet (a "DMZ"), we can economically monitor our greatest potential source of attacks by passively watching the DMZ link. However, the link is an FDDI ring, so to monitor it requires a system that can capture traffic at speeds of up to 100 Mbps. In addition, the volume of traffic over the link is fairly hefty, about 20 GB/day.

No packet filter drops If an application using a packet filter cannot consume packets as quickly as they arrive on the monitored link, then the filter buffers the packets for later consumption. However, eventually the filter will run out of buffer, at which point it *drops* any further packets that arrive. From a security monitoring perspective, drops can completely defeat the monitoring, since the missing packets might contain exactly the interesting traffic that identifies a network intruder. Given our first design requirement—high-speed monitoring—then avoiding packet filter drops becomes another strong requirement.

It is sometimes tempting to dismiss a problem such as packet filter drops with an argument that it is unlikely a traffic spike will occur at the same time as an attack happens to be underway. This argument, however, is completely undermined if we assume that an attacker might, in parallel with a break-in attempt, *attack the monitor itself* (see below).

Real-time notification One of our main dissatisfactions with our initial off-line system was the lengthy delay incurred before detecting an attack. If an attack, or an attempted attack, is detected quickly, then it can be much easier to trace back the attacker (for example, by telephoning the site from which they are coming), minimize damage, prevent further break-ins, and initiate full recording of all of the attacker's network activity. Therefore, one of our requirements for Bro was that it detect attacks in real-time. This is not to discount the enormous utility of keeping extensive, permanent logs of network activity for later analysis. Invariably, when we have suffered a break-in, we turn to these logs for retrospective damage assessment, sometimes searching back a number of months.

Mechanism separate from policy Sound software design often stresses constructing a clear separation between mechanism and policy; done properly, this buys both simplicity and flexibility. The problems faced by our system particularly benefit from separating the two: because we have a fairly high volume of traffic to deal with, we need to be able to easily trade-off at different times how we filter, inspect and respond to different types of traffic. If we hardwired these responses into the system, then these changes would be cumbersome (and error-prone) to make.

Extensible Because there are an enormous number of different network attacks, with who knows how many waiting to be discovered, the system clearly must be designed in order to make it easy to add to it knowledge of new types of attacks. In addition, while our system is a research project, it is at the same time a production system that plays a significant role in our daily security operations. Consequently, we need to be able to upgrade it in small, easily debugged increments.

Avoid simple mistakes Of course, we always want to avoid mistakes. However, here we mean that we particularly desire that the way that a site defines its security policy be both clear and as error-free as possible. (For example, we would not consider expressing the policy in C code as meeting these goals.)

The monitor will be attacked We must assume that attackers will (eventually) have full knowledge of the techniques used by the monitor, and access to its source code, and will use this knowledge in attempts to subvert or overwhelm the monitor so that it fails to detect the attacker's break-in activity. This assumption significantly complicates the design of the monitor; but failing to address it is to build a house of cards.

We do, however, allow one further assumption, namely that *the monitor will only be attacked from one end.* That is, given a network connection between hosts A and B, we assume that at most one of A or B has been compromised and might try to attack the monitor, but not both. This assumption greatly aids in dealing with the problem of attacks on the monitor, since it means that *we can trust one of the endpoints* (though we do not know which).

In addition, we note that this second assumption costs us virtually nothing. If, indeed, both A and B have been compromised, then the attacker can establish intricate covert channels between the two. These can be immeasurably hard to detect, depending on how devious the channel is; that our system fails to do so only means we give up on something extremely difficult anyway.

A final important point concerns the broader context for our monitoring system. Our site is engaged in basic, unclassified research. The consequences of a break-in are usually limited to (potentially significant) expenditure in lost time and re-securing the compromised machines, and perhaps a tarnished public image depending on the subsequent actions of the attackers. Thus, while we very much aim to minimize break-in activity, we do not try to achieve "airtight" security. We instead emphasize monitoring over blocking when possible. Obviously, other sites may have quite different security priorities, which we do not claim to address.

In the remainder of this paper we discuss how the design of Bro attempts to meet these goals and constraints. First, in § 2 we give an overview of the structure of the whole system. § 3 presents the specialized `Bro` language used to express a site's security policy. We turn in § 4 to the details of how the system is currently implemented. § 5 discusses attacks on the monitoring system. § 6 looks at the specialized analysis Bro does for four Internet applications: FTP, Finger, Portmapper, and Telnet. § 7 gives the status of the implementation, a brief assessment of its performance, its availability, and thoughts on future directions. Finally, an Appendix illustrates how the different elements of the system come together for monitoring Finger traffic.

2 Structure of the system

Bro is conceptually divided into an "event engine" that reduces a stream of (filtered) packets to a stream of higher-level network events, and an interpreter for a specialized language that is used to express a site's security policy. More generally, the system is structured in layers, as shown in Figure 1. The lower-most layers process the greatest volume of data, and hence must limit the work performed to a minimum. As we go higher up through the layers, the data stream diminishes, allowing for more processing per data item. This basic design reflects the need to conserve processing as much as possible, in order to meet the goals of monitoring high-speed, large volume traffic flows without dropping packets.

2.1 `libpcap`

From the perspective of the rest of the system, just above the network itself is `libpcap` [MLJ94], the packet-capture library used by `tcpdump` [JLM89]. Using `libpcap` gains significant advantages: it isolates Bro from details of the network link technology (Ethernet, FDDI, SLIP, etc.); it greatly aids in porting Bro to different Unix variants (which also makes it easier to upgrade to faster hardware as it becomes available); and it means that Bro can also operate on `tcpdump` save files, making off-line development and analysis easy.

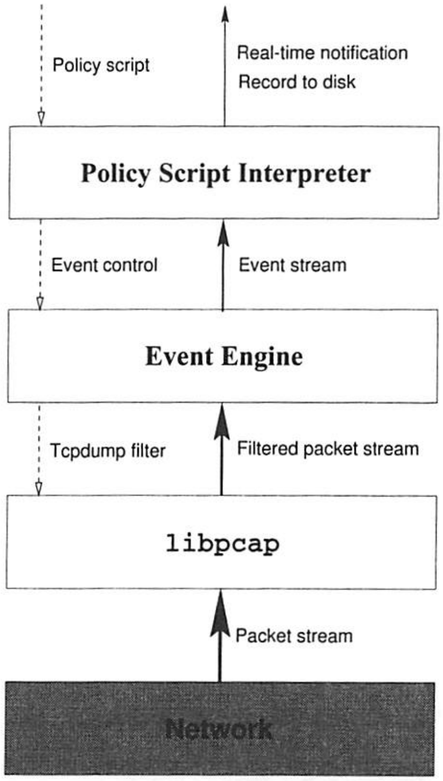


Figure 1: Structure of the Bro system

Another major advantage of `libpcap` is that if the host operating system provides a sufficiently powerful kernel packet filter, such as BPF [MJ93], then `libpcap` downloads the filter used to reduce the traffic into the kernel. Consequently, rather than having to haul every packet up to user-level merely so the majority can be discarded (if the filter accepts only a small proportion of the traffic), the rejected packets can instead be discarded in the kernel, without suffering a context switch or data copying. Winnowing down the packet stream as soon as possible greatly abets monitoring at high speeds without losing packets.

The key to packet filtering is, of course, judicious selection of which packets to keep and which to discard. For the application protocols that Bro knows about, it captures every packet, so it can analyze how the application is being used. In `tcpdump`'s filtering language, this looks like:

```
tcp port finger or tcp port ftp or
tcp port telnet or port 111
```

That is, the filter accepts any TCP packets with a source or destination port of 79 (Finger), 21 (FTP), or 23 (Telnet), and any TCP or UDP packets with a source or destination port of 111 (Portmapper). In addition, Bro uses:

```
    tcp[13] & 7 != 0
```

to capture any TCP packets with the SYN, FIN, or RST control bits set. These packets delimit the beginning (SYN) and end (FIN or RST) of each TCP connection. Because TCP/IP packet headers contain considerable information about each TCP connection, from just these control packets one can extract connection start time, duration, participating hosts, ports (and hence, generally, the name of the application), and the number of bytes sent in each direction. Thus, by capturing on the order of only 4 packets (the two initial SYN packets exchanged, and the final two FIN packets exchanged), we can determine a great deal about a connection even though we filter out all of its data packets.

When using a packet filter, one must also choose a *snapshot length*, which determines how much of each packet should be captured. For example, by default `tcpdump` uses a snapshot length of 68 bytes, which suffices to capture link-layer and TCP/IP headers, but generally discards most of the data in the packet. The smaller the snapshot length, the less data per accepted packet needs to copied up to the user-level by the packet filter, which aids in accelerating packet processing and avoiding loss. On the other hand, to analyze connections at the application level, Bro requires the full data contents of each packet. Consequently, it sets the snapshot length to capture entire packets.

2.2 Event engine

The resulting filtered packet stream is then handed up to the next layer, the Bro "event engine." This layer first performs several integrity checks to assure that the packet headers are well-formed. If these checks fail, then Bro generates an event indicating the problem and discards the packet.

If the checks succeed, then the event engine looks up the connection state associated with the tuple of the two IP addresses and the two TCP or UDP port numbers, creating new state if none already exists. It then dispatches the packet to a handler for the corresponding connection (described shortly). Bro maintains a `tcpdump` trace file associated with the traffic it sees. The connection handler indicates upon return whether the engine should record the entire packet to the trace file, just its header, or nothing at all. This triage trades off the completeness of the traffic trace versus its size and time spent generating the trace. Generally, Bro records full packets if it analyzed the entire packet; just the header if it only analyzed the packet for SYN/FIN/RST computations; and skips recording the packet if it did not do any processing on it.

We now give an overview of general processing done for TCP and UDP packets. In both cases, the processing ends with invoking a handler to process the data payload of the packet. For applications known to Bro, this results in further analysis, as discussed in § 6. For other applications, analysis ends at this point.

TCP processing. For each TCP packet, the connection handler (a C++ virtual function) verifies that the entire TCP header is present and validates the TCP checksum over the packet header and payload. If successful, it then tests whether the TCP header includes any of the SYN/FIN/RST control flags, and if so adjusts the connection's state accordingly. Finally, it processes any data acknowledgement present in the header, and then invokes a handler to process the payload data, if any.

Different changes in the connection's state generate different events. When the initial SYN packet requesting a connection is seen, the event engine schedules a timer for T seconds in the future (presently, five minutes); if the timer expires and the connection has not changed state, then the engine generates a `connection_attempt` event. If before that time, however, the other connection endpoint replies with a correct SYN acknowledgement packet, then the engine immediately generates a `connection_established` event, and cancels the connection attempt timer. On the other hand, if the endpoint replies with a RST packet, then the connection attempt has been rejected, and the engine generates `connection_rejected`. Similarly, if a connection terminates via a normal FIN exchange, then the engine generates `connection_finished`. It also generates several other events reflecting more unusual ways in which connections can terminate.

UDP processing. UDP processing is similar but simpler, since there is no connection state, except in one regard. If host A sends a UDP packet to host B with a source port of p_A and a destination port of p_B, then Bro considers A as having initiated a "request" to B, and establishes pseudo-connection state associated with that request. If B subsequently sends a UDP packet to A with a source port of p_B and destination p_A, then Bro considers this packet to reflect a "reply" to the request. The handlers (virtual functions) for the UDP payload data can then readily distinguish between requests and replies for the usual case when UDP traffic follows that pattern.

The default handlers for UDP requests and replies simply generate udp_request and udp_reply events.

2.3 Policy script interpreter

After the event engine has finished processing a packet, it then checks whether the processing generated any events. (These are kept on a FIFO queue.) If so, it processes each event until the queue is empty, as described below. It also checks whether any timer events have expired, and if so processes them, too.[2]

A key facet of Bro's design is the clear distinction between the generation of events versus what to do in response to the events. These are shown as separate boxes in Figure 1, and this structure reflects the separation between mechanism and policy discussed in § 1. The "policy script interpreter" executes scripts written in the specialized Bro language (detailed in § 3). These scripts specify event handlers, which are essentially identical to Bro functions except that they don't return a value. For each event passed to the interpreter, it retrieves the (semi-)compiled code for the corresponding handler, binds the values of the events to the arguments of the handler, and interprets the code. This code in turn can execute arbitrary Bro scripting commands, including generating new events, logging real-time notifications (using the Unix *syslog* function), recording data to disk, or modifying internal state for access by subsequently invoked event handlers (or by the event engine itself).

Finally, along with separating mechanism from policy, Bro's emphasis on asynchronous events as the link between the event engine and the policy script interpreter buys a great deal in terms of extensibility. Adding new functionality to Bro generally consists of adding a new protocol analyzer to the event engine and then writing new event handlers for the events generated by the analyzer. Neither the analyzer nor the event handlers tend to have much overlap with existing functionality, so for the most part we can avoid the subtle interactions between loosely coupled modules that can easily lead to maintenance headaches and buggy programs.

3 The Bro language

As discussed above, we express security policies in terms of scripts written in the specialized Bro language. In this section we give an overview of the language's features. The aim is to convey the flavor of the language, rather than describe it precisely.

Our goal of "avoid simple mistakes" (§ 1), while perhaps sounding trite, in fact heavily influenced the design of the Bro language. Because intrusion detection can form a cornerstone of the security measures available to a site, we very much want our policy scripts to behave as expected. From our own experience, a big step towards avoiding surprises is to use a strongly typed language that detects typing inconsistencies at compile-time, and that guarantees that all variable references at run-time will be to valid values. Furthermore, we have come to appreciate the benefits of domain-specific languages, that is, languages tailored for a particular task. Having cobbled together our first monitoring system out of tcpdump, awk, and shell scripts, we thirsted for ways to deal directly with hostnames, IP addresses, port numbers, and the like, rather than devising ASCII pseudo-equivalents. By making these sorts of entities first-class values in Bro, we both increase the ease of expression offered by the language and, due to strong typing, catch errors (such as comparing a port to an IP address) that might otherwise slip by.

3.1 Data types and constants

Atomic types. Bro supports several types familiar to users of traditional languages: bool for booleans, int for integers, count for non-negative integers ("unsigned" in C), double for double-precision floating point, and string for a series of bytes. The first four of these (all but string) are termed *arithmetic* types, and mixing them in expressions promotes bool to count, count to int, and int to double.

Bro provides T and F as bool constants for true and false; a series of digits for count constants; and C-style constants for double and string.

Unlike in C, however, Bro strings are represented internally as a count and a vector of bytes, rather than a NUL-terminated series of bytes. This difference is important

[2] There is a subtle design decision involved with processing all of the generated events before proceeding to read the next packet. We might be tempted to defer event processing until a period of relatively light activity, to aid the engine with keeping up during periods of heavy load. However, doing so can lead to races: the "event control" arrow in Figure 1 reflects the fact that the policy script can, to a limited degree, manipulate the connection state maintained inside the engine. If event processing is deferred, then such control may happen after the connection state has already been changed due to more recently-received traffic. So, to ensure that event processing always reflects fresh data, and does not inadvertently lead to inconsistent connection state, we process events immediately, before moving on to newly-arrived network traffic.

because NULs can easily be introduced into strings derived from network traffic, either by the nature of the application, inadvertently, or maliciously by an attacker attempting to subvert the monitor. An example of the latter is sending the following to an FTP server:

```
USER nice\0USER root
```

where "\0" represents a NUL. Depending on how it is written, the FTP application receiving this text might well interpret it as two separate commands, "USER nice" followed by "USER root". But if the monitoring program uses NUL-terminated strings, then it will effectively see only "USER nice" and have no opportunity to detect the subversive action.

Similarly, it is important that when Bro logs such strings, or prints them as text to a file, that it expands embedded NULs into visible escape sequences to flag their appearance.

Bro also includes a number of non-traditional types, geared towards its specific problem domain. A value of type time reflects an absolute time, and interval a difference in time. Subtracting two time values yields an interval; adding or subtracting an interval to a time yields a time; adding two time values is an error. There are presently no time constants, but interval constants can be specified using a numeric (possibly floating-point) value followed by a unit of time, such as "30 min" for thirty minutes.

The port type corresponds to a TCP or UDP port number. TCP and UDP ports are distinct (internally, Bro distinguishes between the two, both of which are 16-bit quantities, by storing port values in a 32-bit integer and setting bit 17 for UDP ports). Thus, a variable of type port can hold either a TCP or a UDP port, but at any given time it is holding exactly one of these.

There are two forms of port constants. The first consists of an unsigned integer followed by either "/tcp" or "/udp." So, for example, "80/tcp" corresponds to TCP port 80 (the HTTP protocol used by the World Wide Web). The second form of constant is specified using an identifier that matches one of the services known to the *getservbyname* library routine. (Probably these service names should instead be built directly into Bro, to avoid problems when porting Bro scripts between operating systems.) So, for example, "telnet" is a Bro constant equivalent to "23/tcp."

This second form of port constant, while highly convenient and readable, brings with it a subtle problem. Some names, such as "domain," on many systems correspond to two different ports; in this example, to 53/tcp and 53/udp. Therefore, the type of "domain" is not a simple port value, but instead a list of port values. Accordingly, a constant like "domain" cannot be used in Bro expressions (such as "dst_port == domain"), because it is ambiguous which value is intended. We return to this point shortly.

Values of type port may be compared for equality or ordering (for example, "20/tcp < telnet" yields true), but otherwise cannot be operated on.

Another networking type provided by Bro is addr, corresponding to an IP address. These are represented internally as unsigned, 32-bit integers, but in Bro scripts the only operations that can be performed on them are comparisons for equality or inequality (also, a built-in function provides masking, as discussed below). Constants of type addr have the familiar "dotted quad" format, $A_1.A_2.A_3.A_4$, where the A_i all lie between 0 and 255.

More interesting are *hostname* constants. There is no Bro type corresponding to Internet hostnames, because hostnames can correspond to multiple IP addresses, so one quickly runs into ambiguities if comparing one hostname with another. Bro does, however, support hostnames as constants. Any series of two or more identifiers delimited by dots forms a hostname constant, so, for example, "lbl.gov" and "www.microsoft.com" are both hostname constants (the latter, as of this writing, corresponds to 13 distinct IP addresses). The value of a hostname constant is a list of addr containing one or more elements. These lists (as with the lists associated with certain port constants, discussed above) cannot be used in Bro expressions; but they play a central role in initializing Bro table's and set's, discussed in § 3.3 below.

Aggregate types. Bro also supports a number of aggregate types. A record is a collection of elements of arbitrary type. For example, the predefined conn_id type, used to hold connection identifiers, is defined in the Bro run-time initialization file as:

```
type conn_id: record {
    orig_h: addr;
    orig_p: port;
    resp_h: addr;
    resp_p: port;
};
```

The orig_h and resp_h elements (or "fields") have type addr and hold the connection originator's and responder's IP addresses. Similarly, orig_p and resp_p hold the originator and responder ports. Record fields are accessed using the "$" operator.

For specifying security policies, a particularly useful

Bro type is table. Bro tables have two components, a set of *indices* and a *yield type*. The indices may be of any atomic (non-aggregate) type, and/or any record types that, when (recursively) expanded into all of their elements, are comprised of only atomic types. (Thus, Bro tables provide a form of associative array.) So, for example,

```
table[port] of string
```

can be indexed by a port value, yielding a string, and:

```
table[conn_id] of ftp_session_info
```

is indexed by a conn_id record—or, equivalently, by an addr, a port, another addr, and another port—and yields an ftp_session_info record as a result.

Closely related to table types are set types. These are simply table types that do not yield a value. Their purpose is to maintain collections of tuples, expressed in terms of the set's indices. The examples in § 3.3 clarify how this is useful.

Another aggregate type supported is file. Support for files is presently crude: a script can open files for writing or appending, and can pass the resulting file variable to the print command to specify where it should write, but that is all. Also, these files are simple ASCII. In the future, we plan to extend files to support reading, ASCII parsing, and binary (typed) reading and writing.

We also note that a key type missing from Bro is that of pattern, for supporting regular expression matching against text. We plan to add patterns in the near future.

Finally, above we alluded to the list type, which holds zero or more instances of a value. Currently, this type is not directly available to the Bro script writer, other than implicitly when using port or *hostname* constants. Since its present use is primarily internal to the script interpreter (when initializing variables, per § 3.3), we do not describe it further.

3.2 Operators

Bro provides a number of C-like operators (+, -, *, /, %, !, &&, ||, ?:, relationals like <=) with which we assume the reader is familiar, and will not detail here. Assignment is done using =, table and set indexing with [], and function invocation and event generation with (). Numeric variables can be incremented and decremented using ++ and --. Record fields are accessed using $, to avoid ambiguity with *hostname* constants. Assignment of aggregate values is *shallow*—the newly-assigned variable refers to the same aggregate value as the right-hand side of the assignment expression. This choice was made to facilitate performance; we have not yet been bitten by the semantics (which differ from C). We may in the future add a copy operator to construct "deep" copies.

From the perspective of C, the only novel operators are in and !in. These infix operators yield bool values depending on whether or not a given index is in a given table or set. For example, if sensitive_services is a set indexed by a single port, then

```
23/tcp in sensitive_services
```

returns true if the set has an element corresponding to an index of TCP port 23, false if it does not have such an element. Similarly, if RPC_okay is a set (or table) indexed by a source address, a destination address, and an RPC service number (a count), then

```
[src_addr, dst_addr, serv] in RPC_okay
```

yields true if the given ordered triple is present as an index into RPC_okay. The !in operator simply returns the boolean negation of the in operator.

Presently, indexing a table or set with a value that does not correspond to one of its elements leads to a run-time error, so such operations need to be preceded by in tests. We find this not entirely satisfying, and plan to add a mechanism for optionally specifying the action to take in such cases on a per-table basis.

Finally, Bro includes a number of predefined functions to perform operations not directly available in the language. Some of the more interesting: fmt provides *sprintf*-style formatting for use in printing or manipulating strings; edit returns a copy of a string that has been edited using the given editing characters (currently it only knows about single-character deletions); mask_addr takes an addr and returns another addr corresponding to its top n bits; open and close manipulate files; network_time returns the timestamp of the most recently received packet; getenv provides access to environment variables; skip_further_processing marks a connection as not requiring any further analysis; set_record_packets instructs the event engine whether or not to record any of a connection's future packets (though SYN/FIN/RST are always recorded);

and parse_ftp_port takes an FTP "PORT" command and returns a record with the corresponding addr and port.

3.3 Variables

Bro supports two levels of scoping: local to a function or event handler, and global to the entire Bro script. Experience has already shown that we would benefit by adding a third, intermediate level of scoping, perhaps as part of a "module" or "object" facility, or even as simple as C's static scoping. Local variables are declared using the keywork local, and the declarations must come inside the body of a function or event handler. There is no requirement to declare variables at the beginning of the function. The scope of the variable ranges from the point of declaration to the end of the body. Global variables are declared using the keyword global and the declarations must come outside of any function bodies. For either type of declaration, the keyword can be replaced instead by const, which indicates that the variable's value is constant and cannot be changed.

Syntactically, a variable declaration looks like:

```
{class} {identifier} [':' {type}] ['=' {init}]
```

That is, a class (local or global scope, or the const qualifier), the name of the variable, an optional type, and an optional initialization value. One of the latter two must be specified. If both are, then naturally the type of the initialization much agree with the specified type. If only a type is given, then the variable is marked as not having a value yet; attempting to access its value before first setting it results in a run-time error.

If only an initializer is specified, then Bro infers the variable's type from the form of the initializer. This proves quite convenient, as does the ease with which complex tables and sets can be initialized. For example,

```
const IRC = { 6666/tcp, 6667/tcp, 6668/tcp };
```

infers a type of set[port] for IRC, while:

```
const ftp_serv = { ftp.lbl.gov, www.lbl.gov };
```

infers a type of set[addr] for ftp_serv, and initializes it to consist of the IP addresses for ftp.lbl.gov and www.lbl.gov, which, as noted above, may encompass more than two addresses. Bro infers compound indices by use of [] notation:

```
const allowed_services = {
    [ftp.lbl.gov, ftp], [ftp.lbl.gov, smtp],
    [ftp.lbl.gov, auth], [ftp.lbl.gov, 20/tcp],
    [www.lbl.gov, ftp], [www.lbl.gov, smtp],
    [www.lbl.gov, auth], [www.lbl.gov, 20/tcp],
    [nntp.lbl.gov, nntp]
};
```

results in allowed_services having type set[addr, port]. Here again, the *hostname* constants may result in more than one IP address. Any time Bro encounters a list of values in an initialization, it replicates the corresponding index. Furthermore, one can explicitly introduce lists in initializers by enclosing a series of values (with compatible types) in []'s, so the above could be written:

```
const allowed_services: set[addr, port] = {
    [ftp.lbl.gov, [ftp, smtp, auth, 20/tcp]],
    [www.lbl.gov, [ftp, smtp, auth, 20/tcp]],
    [nntp.lbl.gov, nntp]
};
```

The only cost of such an initialization is that Bro's algorithm for inferring the variable's type from its initializer currently gets confused by these embedded lists, so the type now needs to be explicitly supplied, as shown.

In addition, any previously-defined global variable can be used in the initialization of a subsequent global variable. If the variable used in this fashion is a set, then its indices are expanded as if enclosed in their own list. So the above could be further simplified to:

```
const allowed_services: set[addr, port] = {
    [ftp_serv, [ftp, smtp, auth, 20/tcp]],
    [nntp.lbl.gov, nntp]
};
```

Initializing table values looks very similar, with the difference that a table initializer includes a *yield* value, too. For example:

```
global port_names = {
    [7/tcp] = "echo",
    [9/tcp] = "discard",
    [11/tcp] = "systat",
    ...
};
```

which infers a type of table[port] of string.

We find that these forms of initialization shorthand are much more than syntactic sugar. Because they allow us to define large tables in a succinct fashion, by referring to previously-defined objects and by concisely capturing forms of replication in the table, we can specify intricate policy relationships in a fashion that's both easy to write and easy to verify. Certainly, we would prefer the final definition of allowed_services above to any of its predecessors, in terms of knowing exactly what the set consists of.

Along with clarity and conciseness, another important advantage of Bro's emphasis on tables and sets is speed. Consider the common problem of attempting to determine whether access is allowed to service S of host H. Rather than using (conceptually):

```
if ( H == ftp.lbl.gov || H == www.lbl.gov )
    if ( S == ftp || S == smtp || ... )
else if ( H == nntp.lbl.gov )
    if ( S == nntp )
...
```

we can simply use:

```
if ( [S, H] in allowed_services )
     ... it's okay ...
```

The in operation translates into a single hash table lookup, avoiding the cascaded if's and clearly showing the intent of the test.

3.4 Statements

Bro currently supports only a modest group of statements, which we have so far found sufficient. Along with C-style if and return and expression evaluation, other statements are: print a list of expressions to a file (*stdout* by default); log a list of expressions; add an element to a set; delete an element from a set or a table; and event, which generates a new event.

In particular, the language does not support looping using a for-style construct. We are wary of loops in event handlers because they can lead to arbitrarily large processing delays, which in turn could lead to packet filter drops. We wanted to see whether we could still adequately express security policies in Bro without resorting to loops; if so, then we have some confidence that every event is handled quickly. So far, this experiment has been successful. Looping is still possible via recursion (either functions calling themselves, or event handlers generating their own events), but we have not found a need to resort to it.

Like in C, we can group sets of statements into *blocks* by enclosing them within {}'s. Function definitions look like:

```
function endpoint_id(h: addr, p: port): string
    {
    if ( p in port_names )
       return fmt("%s/%s", h, port_names[p]);
    else
       return fmt("%s/%d", h, p);
    }
```

Event handler definitions look the same except that function is replaced by event and they cannot specify a return type. See Appendix A for an example.

Functions are invoked the usual way, as expressions specified by the function's name followed by its arguments enclosed within parentheses. Events are generated in a similar fashion, except using the keyword event before the handler's name and argument list. Since events do not return values (they can't, since they are processed asynchronously), event generation is a statement in Bro and not an expression.

Bro also allows "global" statements that are not part of a function or event handler definition. These are executed after parsing the full script, and can of course invoke functions or generate events. The event engine also generates events during different phases of its operation: bro_init when it is about to begin operation, bro_done when it is about to terminate, and bro_signal when it receives a Unix signal.

One difference between defining functions and defining event handlers is that Bro allows multiple, different definitions for a given event handler. Whenever an event is generated, each instance of a handler is invoked in turn (in the order they appear in the script). So, for example, different (conceptual) modules can each define bro_init handlers to take care of their initialization. We find this considerably simplifies the task of creating modular sets of event handlers, but we anticipate requiring greater control in the future over the exact order in which Bro invokes multiple handlers.

4 Implementation issues

We implemented the Bro event engine and script interpreter in C++, currently about 22,000 lines. In this section we discuss some of the significant implementation decisions and tradeoffs. We defer to § 5 discussion of how Bro defends against attacks on the monitoring system, and postpone application-specific issues until § 6, as that discussion benefits from notions developed in § 5.

Single-threaded design. Since event handling lies at the heart of the system, it is natural to consider a multi-threaded design, with one thread per active event handler. We have so far resisted this approach, because of concerns that it could lead to subtle race conditions in Bro scripts.

An important consequence of a single-threaded design is that the system must be careful before initiating any activity that may potentially block waiting for a resource, leading to packet filter drops as the engine fails to consume incoming traffic. A particular concern is performing Domain Name System (DNS) lookups, which can take many seconds to complete or time out. Currently, Bro only performs such lookups when parsing its input file, but we want in the future to be able to make address and hostname translations on the fly, both to generate clearer messages, and to detect certain types of attacks. Consequently, Bro includes customized non-blocking DNS routines that perform DNS lookups asynchronously.

We may yet adopt a multi-threaded design. A more likely possibility is evolving Bro towards a distributed design, in which loosely-coupled, multiple Bro's on separate processors monitor the same network link. Each Bro would watch a different type of traffic (e.g., HTTP or NFS) and communicate only at a high level, to convey current threat information.[3]

Managing timers. Bro uses numerous timers internally for operations such as timing out a connection establishment attempt. It sometimes has thousands of timers pending at a given moment. Consequently, it is important that timers be very lightweight: quick to set and to expire. Our initial implementation used a single priority heap, which we found attractive since insert and delete operations both require only $O(\log(N))$ time if the heap contains N elements. However, we found that when the heap grows quite large—such as during a hostile port scan that creates hundreds of new connections each second—then this overhead becomes significant. Consequently, we perceived a need to redesign timers to bring the overhead closer to $O(1)$. To achieve this, Bro is now in the process of being converted to using "calendar queues" instead [Br88].

A related issue with managing timers concerns exactly when to expire timers. Bro derives its notion of time from the timestamps provided by `libpcap` with each packet it delivers. Whenever this clock advances to a time later than the first element on the timer queue, Bro begins removing timers from the queue and processing their expiration, continuing until the queue is empty or its first element has a timestamp later than the current time. This approach is flawed, however, because in some situations—such as port scans—the event engine may find it needs to expire hundreds of timers that have suddenly become due, because the clock has advanced by a large amount due to a lull in incoming traffic. Clearly, what we should do instead is (again) sacrifice exactness as to when timers are expired, and (1) expire at most k for any single advance of the clock, and (2) also expire timers when there has been a processing lull (as this is precisely the time when we have excess CPU cycles available), without waiting for a packet to finally arrive and end the lull. These changes are also part of our current revisions to Bro's timer management.

Interpreting vs. compiling. Presently, Bro interprets the policy script: that is, it parses the script into a tree of C++ objects that reflect an abstract syntax tree (AST), and then executes portions of the tree as needed by invoking a virtual evaluation method at the root of a given subtree. This method in turn recursively invokes evaluation methods on its children.

Such a design has the virtues of simplicity and ease of debugging, but comes at the cost of considerable overhead. From its inception, we intended Bro to readily admit compilation to a low-level virtual machine. Execution profiles of the current implementation indicate that the interpretive overhead is indeed significant, so we anticipate developing a compiler and optimizer. (The current interpreter does some simple constant folding and peephole optimization when building the AST, but no more.)

Using an interpreter also inadvertantly introduced an implementation problem. By structuring the interpreter such that it recursively invokes virtual evaluation methods on the AST, we wind up intricately tying the Bro evaluation stack with the C++ run-time stack. Consequently, we cannot easily bundle up a Bro function's execution state into a closure to execute at some later point in time. Yet we would like to have this functionality, so Bro scripts have timers available to them; the semantics of these timers are to execute a block of statements when a timer expires, including access to the local variables of the function or event handler scheduling the timer. Therefore, adding timers to Bro will require at a minimum implementing an execution stack for Bro scripts separate from that of the interpreter.

Checkpointing. We run Bro continuously to monitor our DMZ network. However, we need to periodically checkpoint its operation, both to reclaim memory tied up in remembering state for long-dormant connections (because we don't yet have timers in the scripting language;

[3] Some systems, such as DIDS and CSM, orchestrate multiple monitors watching multiple network links, in order to track users as they move from machine to machine [MHL94, WFP96]. These differ from what we envision for Bro in that they require each host in the network to run a monitor.

see above), and to collect a snapshot for archiving and off-line analysis (discussed below).

Checkpointing is currently a three-stage process. First, we run a new instance of Bro that parses the policy script and resolves all of the DNS names in it. Because we have non-blocking DNS routines, Bro can perform a large number of lookups in parallel, as well as timing out lookup attempts whenever it chooses. For each lookup, it compares the results with any it may have previously cached and generates corresponding events (mapping valid, mapping unverified if it had to time out the lookup, or mapping changed). It then updates the DNS cache file and exits.

In the second stage, we run another instance of Bro, this time specifying that it should only consult the DNS cache and not perform lookups. Because it works directly out of the cache, it starts very quickly. After waiting a short interval, we then send a signal to the long-running Bro telling it to terminate. When it exits, the checkpointing is complete.

We find the checkpointing deficient in two ways. First, it would be simpler to coordinate a checkpoint if a new instance of Bro could directly signal an old instance to announce that it is ready to take over monitoring. Second, and more important, currently no state survives the checkpointing. In particular, if the older Bro has identified some suspect activity and is watching it particularly closely (say, by recording all of its packets), this information is lost when the new Bro takes over. Clearly, we need to fix this.

Off-line analysis. As mentioned above, one reason for checkpointing the system is to facilitate off-line analysis. The first step of this analysis is to copy the `libpcap` save file and any files generated by the policy script to an analysis machine. Our policy script generates six such files: a summary of all connection activity, including starting time, duration, size in each direction, protocol, endpoints (IP addresses), connection state, and any additional information (such as username, when identified); a summary of the network interface and packet filter statistics; a list of all generated log messages; summaries of Finger and FTP commands; and a list of all unusual networking events.

Regarding this last, the event engine identifies more than 50 different types of unusual behavior, such as incorrect connection initiations and terminations, checksum errors, packet length mismatches, and protocol violations. For each, it generates a `conn_weird` or `net_weird` event, identifying the behavior with a predefined string. Our policy script uses a `table[string] of count` to map these strings to three different values, "ignore," "file," and "log," meaning ignore the behavior entirely, record it to the anomaly file, or log it (real-time notification) and record it to the file. Some anomalies prove surprisingly common, and on a typical day the anomaly file contains on the order of 1,000 entries, even though our script suppresses duplicate messages.

All of the copied files thus form an archival record of the day's traffic. We keep these files indefinitely. They can prove invaluable when we discover a break-in that first occurred weeks or months in the past. In addition, once we have identified an attacking site, we can run it through the archive to find any other hosts it may have attacked that the monitoring failed to detect (quite common, for example, when the attacker has obtained a list of passwords using a password-sniffer).

In addition, after each checkpoint the analysis machine further studies the traffic logs, looking for possible attacks, the most significant being port scans and address sweeps. We intend to eventually move this analysis into the real-time portion of the system; for now, it waits upon adding timers to `Bro` so we can time out connection state and avoiding consuming huge amounts of memory trying to remember every distinct port and address to which each host has connected.

Finally, the off-line analysis generates a traffic summary highlighting the busiest hosts and giving the volume (number of connections and bytes transferred) due to different applications. As of this writing, on a typical day our site engages in about 600,000 connections transferring 20 GB of data. The great majority (75–80%) of the connections are HTTP; the highest byte volume comes from HTTP, FTP data, NNTP (network news), and, sometimes, X11, with the ordering among them variable.

5 Attacks on the monitor

In this section we discuss the difficult problem of defending the monitor against attacks upon itself. We defer discussion of Bro's application-specific processing until after this section, because elements of that processing reflect attempts to defeat the types of attacks we describe here.

As discussed in § 1, we assume that such attackers have full access to the monitor's algorithms and source code; but also that they have control over only one of the two connection endpoints. In addition, we assume that the cracker does *not* have access to the `Bro` policy script, which each site will have customized, and should keep well protected.

While previous work has addressed the general problem of testing intrusion detection systems [PZCMO96], this work has focussed on correctness of the system in terms of whether it does indeed recognize the attacks claimed. To our knowledge, the literature does not contain any discussion of attacks specifically aimed at subverting a network intrusion detection system, other than the discussion in [PZCMO96] of the general problem of the monitor failing to keep up due to high load.

For our purposes, we classify network monitor attacks into three categories: *overload*, *crash*, and *subterfuge*. The remainder of this section defines each category and briefly discusses the degree to which Bro meets that class of threat.

5.1 Overload attacks

We term an attack as an *overload* if the goal of the attack is to overburden the monitor to the point where it fails to keep up with the data stream it must process. The attack has two phases, the first in which the attacker drives the monitor to the point of overload, and the second in which the attacker attempts a network intrusion. The monitor would ordinarily detect this second phase, but fails to do so—or at least fails to do so with some non-negligible probability—because it is no longer tracking all of the data necessary to detect every current threat.

It is this last consideration, that the attack might still be detected because the monitor was not sufficiently overwhelmed, that complicates the use of overload attacks; so, in turn, this provides a defensive strategy, namely to leave some doubt as to the exact power and typical load of the monitor.

Another defensive strategy is for the monitor to *shed load* when it becomes unduly stressed (see [CT94] for a discussion of shedding load in a different context). For example, the monitor might decide to cease to capture HTTP packets, as these form a high proportion of the traffic. Of course, if the attacker knows the form of load-shedding used by the monitor, then they can exploit its consequent blindness and launch a now-undetected attack.

For Bro in particular, to develop an overload attack one might begin by inspecting Figure 1 to see how to increase the data flow. One step is to send packets that match the packet filter; another, packet streams that in turn generate events; and a third, events that lead to logging or recording to disk.

The first of these is particularly easy, because the `libpcap` filter used by Bro is fixed. One defense against it is to use a hardware platform with sufficient processing power to keep up with a high volume of filtered traffic, and it was this consideration that lead to our elaborating the goal of "no packet filter drops" in § 1. The second level of attack, causing the engine to generate a large volume of events, is a bit more difficult to achieve because Bro events are designed to be lightweight. It is only the events for which the policy specifies quite a bit of work that provide much leverage for an attack at this level, and we do *not* assume that the attacker has access to the policy scripts. This same consideration makes an attack at the final level—elevating the logging or recording rate—difficult, because the attacker does not necessarily know which events lead to logging.

Finally, to help defend against overload attacks, the event engine generates a `net_stats_update` event every T seconds. The value of this event gives the number of packets received, the number dropped by the packet filter due to insufficient buffer, and the number reported dropped by the network interface because the kernel failed to consume them quickly enough. Thus, `Bro` scripts at least have some basic information available to them to determine whether the monitor is becoming overloaded.

5.2 Crash attacks

Crash attacks aim to knock the monitor completely out of action by causing it to either fault or run out of resources. As with an overload attack, the crash attack has two phases, the first during which the attacker crashes the monitor, and the second during which they then proceed with an intrusion.

Crash attacks can be much more subtle than overload attacks, though. By careful source code analysis, it may be possible to find a series of packets, or even just one, that, when received by the monitor, causes it to fault due to a coding error. The effect can be immediate and violent.

We can perhaps defend against this form of crash attack by careful coding and testing. Another type of crash attack, harder to defend against, is one that causes the monitor to exhaust its available resources: dynamic memory or disk space. Even if the monitor has no memory leaks, it still needs to maintain state for any active traffic. Therefore, one attack is to create traffic that consumes a large amount of state. When `Bro` supports timers for policy scripts, this attack will become more difficult, because it will be harder to predict the necessary level of bogus traffic. Attacks on disk space are likewise difficult, unless one knows the available disk capacity. In addition, the monitor might continue to run even with no disk space available, sacrificing an archival record but still producing real-time notifications, so a disk space attack might fail to mask a follow-on attack.

Bro provides two features to aid with defending against crash attacks. First, the event engine maintains a "watchdog" timer that expires every T seconds. (This timer is not a Bro internal timer, but rather a Unix "alarm.") Upon expiration, the watchdog handler checks to see whether the event engine has failed to finish processing the packet (and subsequent events) it was working on T seconds before. If so, then the watchdog presumes that the engine is in some sort of processing jam (perhaps due to a coding error, perhaps due to excessive time spent managing overburdened resources), and terminates the monitor process (first logging this fact, of course, and generating a core image for later analysis).

This feature might not seem particularly useful, except for the fact that it is coupled with a second feature: the script that runs Bro also detects if it ever unduly exits, and, if so, logs this fact and then executes a copy of `tcpdump` that records the same traffic that the monitor would have captured. Thus, crash attacks are (1) logged, and (2) do not allow a subsequent intrusion attempt to go unrecorded, only to evade real-time detection. However, there is a window of opportunity between the time when the Bro monitor crashes and when `tcpdump` runs. If an attacker can predict exactly when this window occurs, then they can still evade detection. But determining the window is difficult without knowledge of the exact configuration of the monitoring system.

5.3 Subterfuge attacks

In a *subterfuge* attack, an attacker attempts to mislead the monitor as to the meaning of the traffic it analyzes. These attacks are particularly difficult to defend against, because (1) unlike overload and crash attacks, if successful they do not leave any traces that they have occurred, and (2) the attacks can be quite subtle. Access to the monitor's source code particularly aids with devising subterfuge attacks.

We briefly discussed an example of a subterfuge attack in § 3.1, in which the attacker sends text with an embedded NUL in the hope that the monitor will miss the text after the NUL. Another form of subterfuge attack is using fragmented IP datagrams in an attempt to elude monitors that fail to reassemble IP fragments (an attack well-known to the firewall community). The key principle is to find a traffic pattern interpreted by the monitor in a different fashion than by the receiving endpoint.

To thwart subterfuge attacks, as we developed Bro we attempted at each stage to analyze the explicit and implicit assumptions made by the system, and how, by violating them, an attack might successfully elude detection. This can be a difficult process, though, and we make no claims to have found them all! In the remainder of this section, we focus on subterfuge attacks on the integrity of the byte stream monitored for a TCP connection. Then, in § 6.4, we look at subterfuge attacks aimed at hiding keywords in interactive text.

To analyze a TCP connection at the application level requires extracting the payload data from each TCP packet and reassembling it into its proper sequence. We now consider a spectrum of approaches to this problem, ranging from simplest and easiest to defeat, to increasingly resilient.

Scanning the data in individual packets without remembering any connection state, while easiest, obviously suffers from major problems: any time the text of interest happens to straddle the boundary between the end of one packet and the beginning of the next, the text will go unobserved. Such a split can happen simply by accident, and certainly by malicious intent.

Some systems address this problem by remembering previously-seen text up to a certain degree (perhaps from the beginning of the current line). This approach fails as soon as a sequence "hole" appears: that is, any time a packet is missing—due to loss or out-of-order delivery—then the resulting discontinuity in the data stream again can mask the presence of key text only partially present.

The next step is to fully reassemble the TCP data stream, based on the sequence numbers associated with each packet. Doing so requires maintaining a list of contiguous data blocks received so far, and fitting the data from new packets into the blocks, merging now-adjacent blocks when possible. At any given moment, one can then scan the text from the beginning of the connection to the highest in-sequence byte received.

Unless we are careful, even keeping track of non-contiguous data blocks does not suffice to prevent a TCP subterfuge attack. The key observation is that an attacker can manipulate the packets their TCP sends so that the monitor sees a particular packet, but the endpoint does not. One way of doing so is to transmit the packet with an invalid TCP checksum. (This particular attack can be dealt with by checksumming every packet, and discarding those that fail; a monitor needs to do this anyway so that it correctly tracks the endpoint's state in the presence of honest data corruption errors, which are not particularly rare [Pa97].) Another way is to launch the packet with an IP "Time To Live" (TTL) field sufficient to carry the packet past the monitoring point, but insufficient to carry it all the way to the endpoint. (If the site has a complex topology, it may be difficult for the monitor to detect this attack.) A third way becomes possible if the final path to the attacked endpoint happens to have a smaller Maximum Transmission Unit (MTU) than the Internet path from the attacker's host to the monitoring point. The attacker then sends a packet with a size exceeding this MTU and with the IP "Don't Fragment" header bit set. This packet will then transit past the monitoring point, but be discarded by the router at the point where the MTU narrows.

By manipulating packets in this fashion, an attacker can send innocuous text for the benefit of the monitor, such as "`USER nice`", and then retransmit (using the same sequence numbers) attack text ("`USER root`"), this time allowing the packets to traverse all the way to the endpoint. If the monitor simply discards retransmitted data without inspecting it, then it will mistakenly believe that the endpoint received the innocuous text, and fail to detect the attack.

A defense against this attack is that when we observe a retransmitted packet (one with data that wholly or partially overlaps previously-seen data), we compare it with any data it overlaps, and sound an alarm (or, for Bro, generate an event) if they disagree. A properly-functioning TCP will always retransmit the same data as originally sent, so any disagreement is either due to a broken TCP (unfortunately, we have observed some of these), undetected data corruption (i.e., corruption the checksum fails to catch), or an attack.

We have argued that the monitor must retain a record of previously transmitted data, both in-sequence and out-of-sequence. The question now arises as to how long the monitor must keep this data around. If it keeps it for the lifetime of the connection, then it may require prodigious amounts of memory any time it happens upon a particularly large connection; these are not infrequent [Pa94]. We instead would like to discard data blocks as soon as possible, to reclaim the associated memory. Clearly, we cannot safely discard blocks above a sequencing hole, as we then lose the opportunity to scan the text that crosses from the sequence hole into the block. But we would like to determine when it is safe to discard in-sequence data.

Here we can make use of our assumption that the attacker controls only one of the connection endpoints. Suppose the stream of interest flows from host A to host B. If the attacker controls B, then they are unable to manipulate the data packets in a subterfuge attack, so we can safely discard the data once it is in-sequence and we have had an opportunity to analyze it. On the other hand, if they control A, then, from our assumption, any traffic we see from B reflects the correct functioning of its TCP (this assumes that we use anti-spoofing filters so that the attacker cannot forge bogus traffic purportedly coming from B). In particular, we can trust that if we see an acknowledgement from B for sequence number n, then indeed B has received all data in sequence up to n. At this point, B's TCP will deliver, or has already delivered, this data to the application running on B. In particular, B's TCP cannot accept any retransmitted data below sequence n, as it has already indicated it has no more interest in such data. Therefore, when the monitor sees an acknowledgement for n, it can safely release any memory associated with data up to sequence n.

6 Application-specific processing

We finish our overview of Bro with a discussion of the additional processing it does for the four applications it currently knows about: Finger, FTP, Portmapper, and Telnet. Admittedly these are just a small portion of the different Internet applications used in attacks, and Bro's effectiveness will benefit greatly as more are added. Fortunately, we have in general found that the system meets our goal of extensibility (§ 1), and adding new applications to Bro is—other than the sometimes major headache of robustly interpreting the application protocol itself—quite straight-forward, a matter of deriving a C++ class to analyze each connection's traffic, and devising a set of events corresponding to significant elements of the application.

6.1 Finger

The first of the applications is the Finger "User Information" service [Zi91]. Structurally, Finger is very simple: the connection originator sends a single line, terminated by a carriage-return line-feed, specifying the user for which they request information. An optional flag requests "full" (verbose) output. The responder returns whatever information it deems appropriate in multiple lines of text, after which it closes the connection.

Bro generates a `finger_request` event whenever it monitors a complete Finger request. A handler for this event looks like:

```
event finger_request(c: connection,
                user: string, full: bool)
```

Our site's policy for Finger requests includes testing for possible buffer-overflow attacks and checking the user against a list of sensitive user ID's, such as privileged accounts. See Appendix A for a discussion of how the Finger analysis is integrated into Bro.

Bro currently does not generate an analogous `finger_reply` event. Two reasons for this are (1) we view the primary threat of Finger to come from the originator and not the responder, so adding `finger_reply` has had a lower priority, and (2) manipulating multi-line strings in `Bro` is clumsy at present, because `Bro` does not have an iteration operator for easily moving through a `table[count] of string`.

A final note: if the event engine finds that the policy script does not define a `finger_request` handler, then it does not bother creating Finger-specific analyzers for new Finger connections. In general, the event engine tries to determine as early as possible whether the user has defined a particular handler, and, if not, avoids undertaking the work associated with generating the corresponding event.

6.2 FTP

The File Transfer Protocol [PR85] is much more complex than the Finger protocol; it also, however, is highly structured and easy to parse, so interpreting an FTP dialog is straight-forward.

For FTP requests, Bro parses each line sent by the connection originator into a command (first word) and an argument (the remainder), splitting the two at the first instance of whitespace it finds, and converting the command to uppercase (to circumvent problems such as a policy script testing for "store file" commands as `STOR` or `stor`, and an attacker instead sending `stOR`, which the remote FTP server will happily accept). It then generates an `ftp_request` event with these and the corresponding connection as arguments.

FTP replies begin with a status code (a number), followed by any accompanying text. Replies also can indicate whether they continue to another line. Accordingly, for each line of reply the event engine generates an `ftp_reply` with the code, the text, a flag indicating continuation, and the corresponding connection as arguments.

As far as the event engine is concerned, that's it—100 lines of straight-forward C++. What is interesting about FTP is that all the remaining work can be done in `Bro` (about 300 lines for our site). The `ftp_request` handler keeps track of distinct FTP sessions, pulls out usernames to test against a list of sensitive ID's (and to annotate the connection's general summary), and, for any FTP request that manipulates a file, checks for access to sensitive files. Some of these checks depend on context; for example, a guest (or "anonymous") user should not attempt to manipulate user-configuration files, while for other users doing so is fine.

A final analysis step for `ftp_request` events is to parse any `PORT` request to extract the hostname and TCP port associated with an upcoming transfer. (The FTP protocol uses multiple TCP connections, one for the control information such as user requests, and others, dynamically created, for each data transfer.) This is an important step, because it enables the script to tell which subsequent connections belong to this FTP session and which do not. A site's policy might allow FTP access to particular servers, but any other access to those servers merits an alarm; but without parsing the `PORT` request, it can be impossible to distinguish a legitimate FTP data transfer connection from an illicit, non-FTP connection. Consequently, the script keeps track of pending data transfer connections, and when it encounters them, marks them as `ftp-data` applications, even if they do not use the well-known port associated with such transfers (the standard does not require them to do so).

We also note that, in addition to correctly identifying FTP-related traffic, parsing `PORT` requests makes it possible to detect "FTP bounce" attacks. In these attacks, a malicious FTP client instructs an FTP server to open a data transfer connection not back to it, but to a third, victim site. The client can thus manipulate the server into uploading data to an arbitrary service on the victim site, or to effectively port-scan the victim site (which the client does by using multiple bogus `PORT` requests and observing the completion status of subsequent data-transfer requests). Our script flags `PORT` requests that attempt any redirection of the data transfer connection. Interestingly, we added this check mostly because it was easy to do so; months later, we monitored the first of several subsequent FTP bounce attacks.

For `ftp_reply` events, most of the work is simply formatting a succinct one-line summary of the request and its result for recording in the FTP activity log. In addition, an FTP `PASV` request has a structure similar to a `PORT` request, except that the FTP server instead of the client determines the specifics of the subsequent data transfer connection. Consequently our script subjects `PASV` replies to the same analysis as `PORT` requests. Finally, there is nothing to prevent a *different* remote host from connecting to the data transfer port offered by a server via a PASV reply. It is hard to see why this might actually occur, but putting in a test for it is simple (unfortunately, there are some false alarms due to multi-homed clients; we use heuristics to reduce these); and, indeed, several months after adding it, it triggered, due to an attacker using 3-way FTP as (evidently) a way to disguise their trail.

6.3 Portmapper

Many services based on Remote Procedure Call (RPC; defined in [Sr95a]) do not listen for requests on a "well-known" port, but rather pick an arbitrary port when initialized. They then register this port with a Portmapper service running on the same machine. Only the Portmapper needs to run on a well-known port; when clients want access to the service, they first contact the Portmapper, and it tells them which port they should then contact in order to reach the service. This second port may be for TCP or UDP access (depending on which the client requests from the Portmapper).

Thus, by monitoring Portmapper traffic, we can detect any attempted access to a number of sensitive RPC services, such as NFS and YP, except in cases where the attacker learns the port for those services some other way (e.g., port-scanning).

The Portmapper service is itself built on top of RPC, which in turn uses the XDR External Data Representation Standard [Sr95b]. Furthermore, one can use RPC on top of either TCP or UDP, and typically the Portmapper listens on both a well-known TCP port and a well-known UDP port (both are port 111). Consequently, adding Portmapper analysis to Bro required adding a generic RPC analyzer, TCP- and UDP-specific analyzers to unwrap the different ways in which RPCs are embedded in TCP and UDP packets, an XDR analyzer, and a Portmapper-specific analyzer.

This last generates six pairs of events, one for each request and reply for the six actions the Portmapper supports: a null call; add a binding between a service and a port; remove a binding; look up a binding; dump the entire table of bindings; and both look up a service and call it directly without requiring a second connection. (This last is a monitoring headache because it means any RPC service can potentially be accessed directly through a Portmapper connection.)

Our policy script for Portmapper traffic again is fairly large, more than 200 lines. Most of this concerns what Portmapper requests we allow between which pairs of hosts, particularly for NFS access.

6.4 Telnet

The final application currently built into Bro is Telnet, a service for remote interactive access [PR83a]. There are several significant difficulties with monitoring Telnet traffic. The first is that, unlike FTP, Telnet traffic is virtually unstructured. There are no nice "USER xyz" directives that make it trivial to identify the account associated with the activity; instead, one must employ a series of heuristics and hope for the best. This problem makes Telnet particularly susceptible to subterfuge attacks, since if the heuristics have holes, an attacker can slip through them undetected.

Our present goal is to determine Telnet usernames in a robust fashion, which we discuss in the remainder of this section. Scanning Telnet sessions for strings reflecting questionable activity is of course also highly interesting, but must wait for us to first add regular expression matching to Bro.

Recognizing the authentication dialog. The first facet of analyzing Telnet activity is to accurately track the initial authentication dialog and extract from it the usernames associated with both login failures and successes. Initially we attempted to build a state machine that would track the various authentication steps: waiting for the username, scanning the login prompt (this comes after the username, since the processing is line-oriented, and the full, newline-terminated prompt line does not appear until after the username has been entered), waiting for the password, scanning the password prompt, and then looking for an indication that the password was accepted or rejected (in which case the process repeats). This approach, though, founders on the great variety of authentication dialogs used by different operating systems, some of which sometimes do not prompt for passwords, or re-prompt for passwords rather than login names after a password failure, or utilize two steps of password authentication, or extract usernames from environment variables, and so on. We now are working on a simpler approach, based on associating particular strings (such as "Password:") with particular information, and not attempting to track the authentication states explicitly. It appears to work better, and its workings are certainly easier to follow.

The Telnet analyzer generates `telnet_logged_in` upon determining that a user has successfully authenticated, `telnet_failure` when a user has failed to authenticate, `authentication_skipped` if it recognizes the authentication dialog as one specified by the policy script as not requiring further analysis, and `telnet_confused` if the analyzer becomes confused regarding the authentication dialog. (This last could, for example, trigger full-packet recording of the subsequent session, for later manual analysis.)

Type-ahead. A basic difficulty that complicates the analysis is type-ahead. We cannot rely on the most-recently entered string as corresponding to the current prompt line. Instead, we keep track of user input lines separately, and consume them as we observe different prompts. For example, if the analyzer scans "Password:", then it associates with the prompt the first unread line in the user type-ahead buffer, and consumes that line. The hazard of this approach is if the Telnet server ever flushes the type-ahead buffer (due to part of its authentication dialog, or upon an explicit signal from the user), then if the monitor misses this fact it will become out of sync. This opens the monitor to a subterfuge attack, in which an attacker passes off an innocuous string as a username, and the policy script in turn fails to recognize that the attacker in fact has authenticated as a privileged user. One fix to this problem—reflecting a strategy we adopt for the more general "keystroke editing" problem discussed below—is to test *both* usernames and passwords against any list of sensitive usernames.

Unless we are careful, type-ahead also opens the door to another subterfuge attack. For example, an attacker can type-ahead the string "Password:", which, when echoed by the Telnet server, would be interpreted by the analyzer as corresponding to a password prompt, when in fact the dialog is in a different state. The analyzer defends against these attacks by checking each typed-ahead string against the different dialog strings it knows about, generating `possible_telnet_ploy` upon a match.

Keystroke editing. Usernames can also become disguised due to use of keystroke editing. For example, we would like to recognize that "`rb`*<DEL>*`oot`" does indeed correspond to a username of `root`, assuming that *<DEL>* is the single-character deletion operator. We find this assumption, however, problematic, since some systems use *<DEL>* and others use *<BS>*. We address this problem by applying both forms of editing to usernames, yielding possibly three different strings, each of which the script then assesses in turn. So, for example, the string "`rob`*<DEL><BS><BS>*`ot`" is tested both directly, as "`ro`*<BS><BS>*`ot`", and as "`root`".

Editing is not limited to deleting individual characters, however. Some systems support deleting entire words or lines; others allow access to previously-typed lines using an escape sequence. Word and line deletion do not allow an attacker to hide their username, if tests for sensitive usernames check for any embedded occurrence of the username within the input text. "History" access to previous text is more problematic; presently, the analyzer recognizes the operating system that supports this (VMS) and, for it only, expands the escape sequence into the text of the previous line.

Telnet options. The Telnet protocol supports a rich, complex mechanism for exchanging options between the client and server [PR83b] (there are more than 50 RFCs discussing different Telnet options). Unhappily, we cannot ignore the possible presence of these options in our analysis, because an attacker can embed one in the middle of text they transmit in order to disguise their intent—for example, "`ro`*<option>*`ot`". The Telnet server will dutifully strip out the option before passing along the remaining text to the authentication system. We must do the same. On the other hand, parsing options also yields some benefits: we can detect connections that successfully negotiate to encrypt the data session, and skip subsequent analysis (rather than generating `telnet_confused` events), as well as analyzing options used for authentication (for example, Kerberos) and to transmit the user's environment variables (some systems use `$USER` as the default username during subsequent authentication).

7 Status, performance, and future directions

Bro has operated continuously since April 1996 as an integral part of our site's security system. It initially included only general TCP/IP analysis; as time permitted, we added the additional modules discussed in § 6, and we plan to add many more.

Presently, the implementation is about 22,000 lines of C++ and another 1,900 lines of `Bro` (about 1,650 lines of which are "boilerplate" not specific to our site). It runs under Digital Unix, FreeBSD, IRIX, SunOS, and Solaris operating systems. It generates about 40 MB of connection summaries each day, and an average of 20 real-time notifications, though this figure varies greatly. While most of the notifications are innocuous (and if we were not also developers of the system, we would suppress these), we not infrequently also detect break-in attempts. Operation of the system has resulted so far in 4,000 email messages, 85 incident reports filed with CIAC and CERT, a number of accounts deactivated by other sites, and a couple incidents involving law enforcement.

The system generally operates without incurring any packet drops. The FDDI ring it runs on is heavily used: a recent trace of a 2-3PM busy hour reflects a traffic level of over 17,000 packets/sec (50 Mbps) sustained for the full hour, with peaks exceeding 30,000 packets/sec. However, the packet filter discards a great deal of this, both due to filtering primarily on SYN, FIN, or RST control bits, and because only about 20% of the traffic belongs to networks that we routinely monitor (the link is shared with a large neighbor institution). During a busy hour, the monitor may receive 300,000 packets matching the filter, with peaks of 200/sec.

In order to make a preliminary assessment of the system under stress, we ran it for a thirty-minute period without the "interesting networks" filter, resulting in a much higher fraction of traffic accepted by the packet filter. During this period, the filter accepted an average of 640 packets/sec, with peaks over 800/sec. However, the filter also reported 364 dropped packets. (We note that the hardware platform used is no longer state-of-the-art.)

In addition to developing more application analysis modules, we see a number of avenues for future work. As discussed above, compiling Bro scripts and devising mechanisms to distribute monitoring across multiple CPUs have high priority. We are also very interested in extending BPF to better support monitoring, such as adding lookup tables and variable-length snapshots. Another interesting direction is to add some "teeth" to the monitoring in the form of actively terminating misbehaving connections by sending RST packets to their endpoints, or communicating with intermediary routers. This form of "reactive firewall" might fit particularly well to environments like ours that need to strike a balance between security and openness.

Finally, Bro is publicly available in source-code form (see *http://www-nrg.ee.lbl.gov/bro-info.html* for release information). We hope that it will both benefit the community and in turn benefit from community efforts to enhance it.

8 Acknowledgements

We gratefully acknowledge Digital Equipment Corporation's Western Research Laboratory for contributing the Alpha system that made developing and operating Bro at high speeds possible. I would particularly like to thank Jeff Mogul, who was instrumental in arranging this through WRL's External Research Program.

Many thanks, too, to Craig Leres. Bro has benefited greatly from many discussions with him. Craig also wrote the calendar queue and non-blocking DNS routines discussed in § 4.

My appreciation to Scott Denton and John Antonishek for alpha-testing Bro and contributing portability fixes and other enhancements.

Finally, this work would not have been possible without the support and enthusiasm of Mark Rosenberg, Van Jacobson, Stu Loken and Dave Stevens—much appreciated!

A Example: tracking Finger traffic

In this appendix we give an overview of how the different elements of Bro come together for monitoring Finger traffic. For the event engine, we have a C++ class FingerConn, derived from the general-purpose TCP_Connection class. When Bro encounters a new connection with service port 79, it instantiates a corresponding FingerConn object, instead of a TCP_Connection object as it would for an unrecognized port.

FingerConn redefines the virtual function BuildEndpoints, which is invoked when a connection object is first created:

```
void FingerConn::BuildEndpoints()
   {
   resp = new TCP_Endpoint(this, 0);
   orig = new TCP_EndpointLine(this, 1, 1, 0);
   }
```

Here, resp, corresponding to the responder (Finger server) side of the connection, is initialized to an ordinary TCP_Endpoint object, because Bro does not (presently) look inside Finger replies. But orig, the Finger client side, is initialized to a TCP_EndpointLine object, which means Bro will track the contents of that side of the connection, and, furthermore, deliver the contents in a line-oriented fashion to FingerConn's virtual NewLine function:

```
int FingerConn::NewLine(TCP_Endpoint* /* s */,
                        double /* t */, char* line)
   {
   line = skip_whitespace(line);

   // Check for /W.
   int is_long = (line[0] == '/' &&
                  toupper(line[1]) == 'W');
   if ( is_long )
      line = skip_whitespace(line+2);

   val_list* vl = new val_list;
   vl->append(BuildConnVal());
   vl->append(new StringVal(line));
   vl->append(new Val(is_long, TYPE_BOOL));
```

```
    mgr.QueueEvent(finger_request, vl);
    return 0;
    }
```

NewLine skips whitespace in the request, scans it for the "/W" indicator (which requests verbose Finger output), and moves past it if present. It then creates a val_list object, which holds a list of generic Bro Val objects. The first of these is assigned to a generic connection-identifier value (see below); the second, to a Bro string containing the Finger request, and the third to a bool indicating whether the request was verbose or not. The penultimate line queues a new finger_request event with the corresponding list of values as arguments; finally, return 0 indicates that the FingerConn is all done with the memory associated with line (since new StringVal(line) made a copy of it), so that memory can be reclaimed by the caller.

The connection identifier discussed above is defined in Bro as a "connection" record:

```
type endpoint: record {
   size: count; state: count;
};
type connection: record {
   id: conn_id;
   orig: endpoint; resp: endpoint;
   start_time: time;
   duration: interval;
   service: string;
      # if empty, service not yet determined
   addl: string;
   hot: count;
      # how hot; 0 = don't know or not hot
};
```

The id field is a conn_id record, discussed in § 3.1. orig and resp correspond to the connection originator and responder, each a Bro endpoint record consisting of size (the number of bytes transferred by that endpoint so far) and state, the endpoint's TCP state (e.g., SYN sent, established, closed). This latter would be better expressed using an enumerated type (rather than a count), which we may add to Bro in the future.

The start_time field reflects when the connection's first packet was seen, and duration how long the connection has existed. service corresponds to the name of the service, or an empty string if it has not been identified. By convention, addl holds additional information associated with the connection; better than a string here would be some sort of union or generic type, if Bro supported such. Finally, by convention the policy script increments hot whenever it finds something potentially suspicious about the connection.

Here is the corresponding policy script:

```
global hot_names = { "root", "lp", "uucp" };
global finger_log =
   open(getenv("BRO_ID") == "" ?
        "finger.log" :
        fmt("finger.%s", getenv("BRO_ID")));

event finger_request(c:connection,
                     request: string,
                     full: bool)
   {
   if ( byte_len(request) > 80 ) {
      request = fmt("%s...",
                    sub_bytes(request, 1, 80));
      ++c$hot;
   }
   if ( request in hot_names )
      ++c$hot;

   local req = request == "" ?
      "ANY" : fmt("\"%s\"", request);
   if ( c$addl != "" )
      # This is an additional request.
      req = fmt("(%s)", req);
   if ( full )
      req = fmt("%s (/W)", req);

   local msg = fmt("%s > %s %s",
                   c$id$orig_h,
                   c$id$resp_h,
                   req);
   if ( c$hot > 0 )
      log fmt("finger: %s", msg);
   print finger_log,
         fmt("%.6f %s", c$start_time, msg);

   c$addl = c$addl == "" ?
            req : fmt("*%s, %s", c$addl, req);
   }
```

The global hot_names is a Bro set of string. In the next line, finger_log is initialized to a Bro file, either named "finger.log", or, if the BRO_ID environment variable is set, to a name derived from it using the built-in fmt function.

The finger_request event handler follows. It takes three arguments, corresponding to the values added to the val_list above. It first checks whether the request is excessively long, and, if so, truncates it and increments the hot field of the connection's information record. (The Bro built-in functions used here are named in terms of "bytes" rather than "string" because they make no assumptions about NUL-termination of their arguments; in particular, byte_len returns the length of its argument including a final NUL byte, if present.)

Next, the script checks whether the request corresponds to any of the entries in the hot_names set. If so, it again marks the connection as "hot."

We then initialize the local variable req to a quoted version of the request; or, if the request was empty (which in the Finger protocol indicates a request type of "ANY"), then it is changed to "ANY".

The event handler stores the Finger request in the connection record's `addl` field (see below), so the next line checks to see whether this field already contains a request. If so, then we are seeing multiple requests for a single Finger connection. This is not allowed by the Finger protocol, but that doesn't mean we won't see them! In particular, we might imagine a subterfuge attack in which an attacker queries an innocuous name in their first request, and a sensitive name in their second, and depending on how the finger server is written, it may well respond to both.[4] This script will still catch such use, since it fully processes each request; but it needs to be careful to keep the global state corresponding to the connection (in the `addl` field) complete. To do so, it marks additional requests by enclosing them in parentheses, and also prepends an asterisk to the entire `addl` field for each additional request, so that in later visual inspection of the Finger logs these requests immediately stand out.

The `msg` local variable holds the basic description of the Finger request. The `fmt` function knows to format the IP addresses `c$id$orig_h` and `c$id$resp_h` as "dotted quads."

Next, if the connection has been marked as "hot" (either just previously, or perhaps by a completely different event handler), then the script generates a real-time notification. In any case, it also records the request to the `finger_log` file. Finally, it updates the `addl` field to reflect the request (and to flag multiple requests, as discussed above).

Entries in the log file look like:

```
880988813.752829 171.64.15.68 >
                 128.3.253.104 "feng"
880991121.364126 131.243.168.28 >
                 130.132.143.23 "anlin"
880997120.932007 192.84.144.6 >
                 128.3.32.16 ALL
881000846.603872 128.3.9.45 >
                 146.165.7.14 ALL (/W)
881001601.958411 152.66.83.11 >
                 128.3.13.76 "davfor"
```

(though without the lines split after the ">").

The real-time notifications look quite similar, with the keyword "`finger:`" added to avoid ambiguity with other types of real-time notification.

[4] We do indeed see occasional multiple requests. So far, they have all appeared fully innocuous.

References

[Br88] R. Brown, "Calendar Queues: A Fast $O(1)$ Priority Queue Implementation for the Simulation Event Set Problem," *Communications of the ACM*, 31(10), pp. 1220-1227, Oct. 1988.

[CT94] C. Compton and D. Tennenhouse, "Collaborative Load Shedding for Media-Based Applications," *Proc. International Conference on Multimedia Computing and Systems*, Boston, MA, May. 1994.

[In97] Internet Security Systems, Inc., *RealSecure*^{TM}, http://www.iss.net/prod/rs.html, 1997.

[JLM89] V. Jacobson, C. Leres, and S. McCanne, `tcpdump`, available via anonymous ftp to ftp.ee.lbl.gov, Jun. 1989.

[MJ93] S. McCanne and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," *Proc. 1993 Winter USENIX Conference*, San Diego, CA.

[MLJ94] S. McCanne, C. Leres and V. Jacobson, `libpcap`, available via anonymous ftp to ftp.ee.lbl.gov, 1994.

[MHL94] B. Mukherjee, L. Heberlein, and K. Levitt, "Network Intrusion Detection," *IEEE Network*, 8(3), pp. 26-41, May/Jun. 1994.

[Ne97] Network Flight Recorder, Inc., *Network Flight Recorder*, http://www.nfr.com, 1997.

[Pa94] V. Paxson, "Empirically-Derived Analytic Models of Wide-Area TCP Connections," *IEEE/ACM Transactions on Networking*, 2(4), pp. 316-336, Aug. 1994.

[Pa97] V. Paxson, "End-to-End Internet Packet Dynamics," *Proc. SIGCOMM '97*, Cannes, France, Sep. 1997.

[PR83a] J. Postel and J. Reynolds, "Telnet Protocol Specification," RFC 854, Network Information Center, SRI International, Menlo Park, CA, May 1983.

[PR83b] J. Postel and J. Reynolds, "Telnet Option Specifications," RFC 855, Network Information Center, SRI International, Menlo Park, CA, May 1983.

[PR85] J. Postel and J. Reynolds, "File Transfer Protocol (FTP)," RFC 959, Network Information Center, SRI International, Menlo Park, CA, Oct. 1985.

[PZCMO96] N. Puketza, K. Zhang, M. Chung, B. Mukherjee and R. Olsson, "A Methodology for Testing Intrusion Detection Systems," *IEEE Transactions on Software Engineering*, 22(10), pp. 719-729, Oct. 1996.

[RLSSLW97] M. Ranum, K. Landfield, M. Stolarchuk, M. Sienkiewicz, A. Lambeth and E. Wall, "Implementing a generalized tool for network monitoring," *Proc. LISA '97*, USENIX 11th Systems Administration Conference, San Diego, Oct. 1997.

[Sr95a] R. Srinivasan, "RPC: Remote Procedure Call Protocol Specification Version 2," RFC 1831, DDN Network Information Center, Aug. 1995.

[Sr95b] R. Srinivasan, "XDR: External Data Representation Standard," RFC 1832, DDN Network Information Center, Aug. 1995.

[To97] Touch Technologies, Inc., *INTOUCH INSA*, http://www.ttisms.com/tti/nsa_www.html, 1997.

[Wh97] WheelGroup Corporation, *NetRanger*™, http://www.wheelgroup.com, 1997.

[WFP96] G. White, E. Fisch and U. Pooch, "Cooperating Security Managers: A Peer-Based Intrusion Detection System," *IEEE Network*, 10(1), pp. 20-23, Jan./Feb. 1994.

[Zi91] D. Zimmerman, "The Finger User Information Protocol," RFC 1288, Network Information Center, SRI International, Menlo Park, CA, Dec. 1991.

Cryptographic Support for Secure Logs on Untrusted Machines

Bruce Schneier John Kelsey
{schneier,kelsey}@counterpane.com
Counterpane Systems, 101 East Minnehaha Parkway, Minneapolis, MN 55419

Abstract

In many real-world applications, sensitive information must be kept in log files on an untrusted machine. In the event that an attacker captures this machine, we would like to guarantee that he will gain little or no information from the log files and to limit his ability to corrupt the log files. We describe a computationally cheap method for making all log entries generated prior to the logging machine's compromise impossible for the attacker to read, and also impossible to undetectably modify or destroy.

1 Introduction

A Description of the Problem We have an untrusted machine, $\mathcal{U}$, which is not physically secure or sufficiently tamper-resistant to guarantee that it cannot be taken over by some attacker. However, this machine needs to be able to build and maintain a file of audit log entries of some processes, measurements, events, or tasks.

With a minimal amount of interaction with a trusted machine, $\mathcal{T}$, we want to make the strongest security guarantees possible about the authenticity of the log on $\mathcal{U}$. In particular, we do not want an attacker who gains control of $\mathcal{U}$ at time t to be able to read log entries made before time t, and we do not want him to be able to alter or delete log entries made before time t in such a way that his manipulation will be undetected when $\mathcal{U}$ next interacts with $\mathcal{T}$.

It is important to note that $\mathcal{U}$, while "untrusted," isn't generally expected to be compromised. However, we must be able to make strong statements about the security of previously-generated log entries even if $\mathcal{U}$ is compromised.

In systems where the owner of a device is not the same person as the owner of the secrets within the device, it is essential that audit mechanisms be in place to determine if there has been some attempted fraud. These audit mechanisms must survive the attacker's attempts at undetectable manipulation. This is not a system to prevent all possible manipulations of the audit log; this is a system to detect such manipulations after the fact.

Applications for this sort of mechanism abound. Consider $\mathcal{U}$ to be an electronic wallet—a smart card, a calculator-like device, a dongle on a PC, etc.—that contains programs and data protected by some kind of tamper resistance. The tamper resistance is expected to keep most attackers out, but it is not 100% reliable. However, the wallet occasionally interacts with trusted computers ($\mathcal{T}$) in banks. We would like the wallet to keep an audit log of both its actions and data from various sensors designed to respond to tampering attempts. Moreover, we would like this log to survive successful tampering, so that when the wallet is brought in for inspection it will be obvious that the wallet has been tampered with.

There are other examples of systems that could benefit from this protocol. A computer that logs various kinds of network activity needs to have log entries of an attack undeleteable and unalterable, even in the event that an attacker takes over the logging machine over the network.[1] An intrusion-detection system that logs the entry and exit of people into a secured area needs to resist attempts to delete or alter logs, even after the machine on which the logging takes place has been taken over by an attacker. A secure digital camera needs to guarantee the authenticity of pictures taken, even if it is reverse-engineered sometime later [KSH96]. A computer under the control of a marginally-trusted person or entity needs to keep logs that can't be changed "after the fact," despite the intention of the person in control of the machine to "rewrite history" in some way. A computer that is keeping logs of confidential information needs to keep that information confidential even if it is taken over for a time by some attacker. Mobile computing agents could

[1] In [Sto89], Cliff Stoll attached a printer to a network computer for just this purpose.

benefit from the ability to resist alteration of their logs even when they're running under the control of a malicious adversary.

Limits on Useful Solutions A few moments' reflection will reveal that no security measure can protect the audit log entries written *after* an attacker has gained control of $\mathcal{U}$. At that point, $\mathcal{U}$ will write to the log whatever the attacker wants it to write. All that is possible is to refuse the attacker the ability to read, alter, or delete log entries made *before* he compromised the logging machine.

If there is a reliable, high-bandwidth channel constantly available between $\mathcal{T}$ and $\mathcal{U}$, then this problem won't come up. $\mathcal{U}$ will simply encrypt each log entry as it is created and send it to $\mathcal{T}$ over this channel. Once logs are stored by $\mathcal{T}$, we are willing to trust that no attacker can change them.

Finally, no cryptographic method can be used to actually prevent the deletion of log entries: solving that problem requires write-only hardware such as a writable CD-ROM disk, a WORM disk,or a paper printout. The only thing these cryptographichic protocols can do is to guarantee detection of such deletion, and that is assuming $\mathcal{U}$ eventually manages to communicate with $\mathcal{T}$.

These three statements define the limits of useful solutions to this problem. We are able to make strong statements only about log entries $\mathcal{U}$ made before compromise, and a solution to do so is interesting only when there is no communications channel of sufficient reliability, bandwidth, and security to simply continuously store the logs on $\mathcal{T}$.

In essense, this technique is an implementation of an engineering tradeoff between how "online" $\mathcal{U}$ is and how often we expect $\mathcal{U}$ to be compromised. If we expect $\mathcal{U}$ to be compromised very often—once a minute, for example—then we should send log entries to $\mathcal{T}$ at least once or twice every minute; hence $\mathcal{U}$ will need to be online nearly all the time. In many systems, $\mathcal{U}$ is not expected to be compromised nearly as often, and is also not online nearly as continuously. Therefore, we only need $\mathcal{U}$ to communicate log entries to $\mathcal{T}$ infrequently, at some period related to the frequency with which you expect that T may be compromised. The audit log technique in our paper enables this tradeoff. It provides a "knob" that the system architect can adjust based on his judgement of this tradeoff; furthermore, the knob can be adjusted during the operation of the system as expectations of the rate of compromise change.

Organization of This Paper The remainder of this paper is divided into sections as follows: In Section 2 we discuss notation and tools. In Section 3 we present our general scheme. Then, in Section 4 we discuss some extensions and variations on the scheme. Finally, in Section 5 we provide a summary of what we've done and interesting directions for further research in this area.

2 Notation and Tools

In the remainder of this paper, we will use the following notation:

1. ID_x represents a unique identifier string for an entity, x, within this application.
2. $PKE_{PK_x}(K)$ is the public-key encryption, under x's public key, of K, using an algorithm such as RSA [RSA78] or ElGamal [ElG85].
3. $SIGN_{SK_x}(Z)$ is the digital signature, under x's private key, of Z, using an algorithm such as RSA or DSA [NIST94].
4. $E_{K_0}(X)$ is the symmetric encryption of X under key K_0, using an algorithm such as DES [NBS77], IDEA [LMM91], or Blowfish [Sch94].
5. $MAC_{K_0}(X)$ is the symmetric message authentication code (HMAC or NMAC [BCK96]), under key K_0, of X.
6. $hash(X)$ is the one-way hash, using an algorithm such as SHA-1 [NIST93] or RIPE-MD [DBP96], of X.
7. X, Y represents the concatenation of X with Y.

Descriptions of most of these algorithms are in [Sti95, Sch96, MOV97].

Note that all authenticated protocol steps in this paper should include some nonce identifying the specific application, version, protocol, and step. This nonce serves to limit damaging protocol interactions, either accidental or intentional [And95, KSW97]. In our protocols, we will use p to represent this unique step identifier.

Additionally, many of the protocols require the two parties to establish a secure connection, using an authentication and key-agreement protocol that has perfect forward secrecy, such as authenticated Diffie-Hellman. The purpose of this is for the two parties to prove their identity to each other, and to generate a shared secret with which to encrypt subsequent messages in the protocol.

In the remainder of this paper, we will use the following players:

1. $\mathcal{T}$ is the trusted machine. It may typically be thought of as a server in a secure location, though it may wind up being implemented in various ways: a tamper-resistant token, a bank ATM machine, etc.
2. $\mathcal{U}$ is the untrusted machine, on which the log is to be kept.
3. $\mathcal{V}$ is a moderately-trusted verifier, who will be trusted to review certain kinds of records, but not trusted with the ability to change records. Note that not all of our implementations will be able to support $\mathcal{V}$.

In this paper, we assume that $\mathcal{U}$ has both short-term and long-term storage available. The long-term storage will store the audit log, and we assume that it is sufficiently large that filling it up is not a problem. We assume that $\mathcal{U}$ can irretrievably delete information held in short-term memory, and that this is done each time a new key is derived. We also assume that $\mathcal{U}$ has some way of generating random or cryptographically strong pseudorandom values. Finally, we assume the existence of several cryptographic primitives, and a well-understood way to establish a secure connection across an insecure medium. Methodologies for all of these are described in great detail elsewhere: see [Sti95, Sch96, MOV97].

3 A Description of Our Method

Our system leverages the fact that the untrusted machine creating the logfile initially shares a secret key with a trusted verification machine. With this key, we create the logfile.

The security of our logfile comes from four basic facts:

1. The log's authentication key is hashed, using a one-way hash function, immediately after a log entry is written. The new value of the authentication key overwrites and irretrieveably deletes the previous value.
2. Each log entry's encryption key is derived, using a one-way process, from that entry's authentication key. This makes it possible to give encryption keys for individual logs out to partially-trusted users or entities (so that they can decrypt and read entries), without allowing those users or entities to make undetectable changes.
3. Each log entry contains an element in a hash chain that serves to authenticate the values of all previous log entries [HS91, SK97]. It is this value that is actually authenticated, which makes it possible to remotely verify all previous log entries by authenticating a single hash value.
4. Each log entry contains its own permission mask. This permission mask defines roles in a role-based security scheme; partially-trusted users can be given access to only some kinds of entries. Because the encryption keys for each log entry are derived partly from the log entry type, lying about what permissions a given log entry has ensures that the partially-trusted user simply never gets the right key.

3.1 Log Entry Definitions and Construction Rules

All entries in the log file use the same format, and are constructed according to the following procedure:

1. D_j is the data to be entered in the jth log entry of ID_{log}. The specific data format of D is not specified in our scheme: it must merely be something that the reader of the log entries will unambiguously understand, and that can be distinguished in virtually all cases from random gibberish. (If we are dealing with raw binary data here, we may add some structure to the data to make detection of garbled information likely, though this is seldom going to be important.)
2. W_j is the log entry type of the jth log entry. This type serves as a permissions mask for $\mathcal{V}$; $\mathcal{T}$ will be allowed to control which log entry types any particular $\mathcal{V}$ will be allowed to access.
3. A_j is the authentication key for the jth entry in the log. This is the core secret that provides all of this scheme's security. Note that $\mathcal{U}$ must generate a new A_0 before starting the logfile; A_0 can be given to $\mathcal{U}$ by $\mathcal{T}$ at startup, or $\mathcal{U}$ can

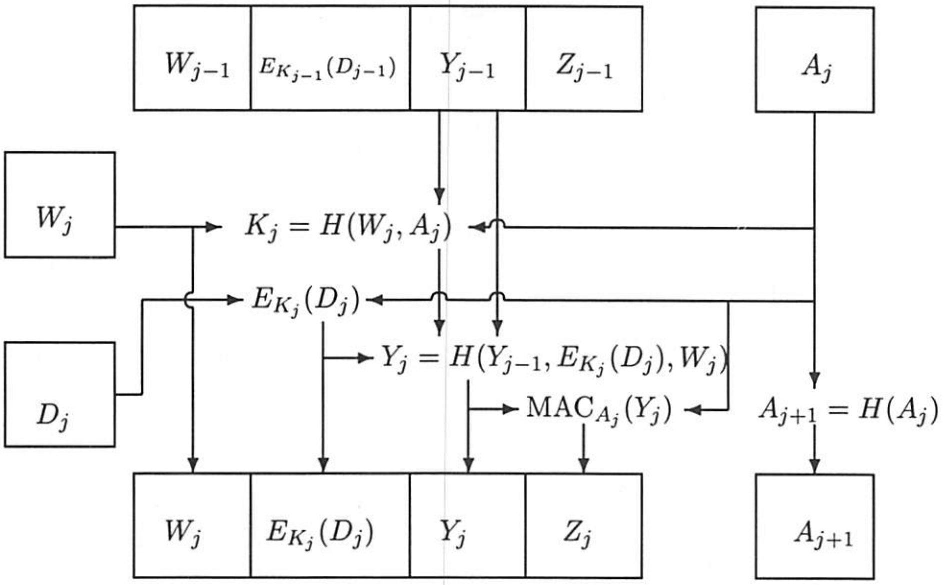


Figure 1: Adding an entry to the log.

randomly generate it and then securely transmit it to $\mathcal{T}$.

4. $K_j = hash(\text{“Encryption Key”}, W_j, A_j)$. This is the key used to encrypt the jth entry in the log. Note that W_j is used in the key derivation to prevent the Verifier getting decryption keys for log entry types to which he is not permitted access.

5. $Y_j = hash(Y_{j-1}, E_{K_j}(D_j), W_j)$. This is the hash chain which we maintain, to allow partially-trusted users, $\mathcal{V}$s, to verify parts of the log over a low-bandwidth connection with the trusted machine, $\mathcal{T}$. Y_j is based on $E_{K_j}(D_j)$ instead of D_j so that the chain can be verified without knowing the log entry. At startup, Y_{-1} is defined as a twenty-byte block of binary zeros.[2]

6. $Z_j = MAC_{A_j}(Y_j)$.

7. $L_j = W_j, E_{K_j}(D_j), Y_j, Z_j$, where L_j is the jth log entry.

8. $A_{j+1} = hash(\text{“Increment Hash”}, A_j)$.

Note that when A_{j+1} and K_j are computed, the previous A_j and K_{j-1} values are irretrieveably destroyed; under normal operation there are no copies of these values kept on $\mathcal{U}$. Additionally, K_j is destroyed immediately after use in Step (4). (Naturally, an attacker will probably store A_j values after he takes control of $\mathcal{U}$.)

The above procedure defines how to write the jth entry into the log, given A_{j-1}, Y_{j-1}, and D_j. See Figure 1 for an illustration of this process.

If an attacker gains control of $\mathcal{U}$ at time t, he will have a list of valid log entries, $L_1, L_2, \ldots, L_t$, and the value A_{t+1}. He cannot compute A_{t-n} for any $n \leq t$, so he cannot read or falsify any previous entry. He can delete a block of entries (or the entire log file), but he cannot create new entries, either past entries to replace them or future entries. The next time $\mathcal{U}$ interacts with $\mathcal{T}$, $\mathcal{T}$ will realize that entries have been deleted from the log and that 1) $\mathcal{U}$ may have committed some invalid actions that have not been properly audited, and 2) $\mathcal{U}$ may have committed some valid actions whose audit record has been deleted. [3]

3.2 Startup: Creating the Logfile

In order to start the logfile, $\mathcal{U}$ must irrevocably commit A_0 to $\mathcal{T}$. Once A_0 has been committed to, there must be a valid log on $\mathcal{U}$, properly formed in all

[2] There is no security reason for this; it has to be initialized as something.

[3] If the attacker gains control of $\mathcal{U}$ before Step (8), he can learn A_t. In this case the tth log entry is not secured from deletion or manipulation.

ways, or $\mathcal{T}$ will suspect that someone is tampering with $\mathcal{U}$.

In the simplest case, $\mathcal{T}$ is able to reliably receive a message (but perhaps not in realtime) from $\mathcal{U}$. $\mathcal{U}$ knows $\mathcal{T}$'s public key, and has a certificate of her own public key from $\mathcal{T}$. The protocol works as follows:

1. $\mathcal{U}$ forms:

   K_0, a random session key.

   d, a current timestamp.

   d^+, timestamp at which $\mathcal{U}$ will time out.

   ID_{log}, a unique identifier for this logfile.

   C_U, $\mathcal{U}$'s certificate from $\mathcal{T}$.

   A_0, a random starting point.

   $X_0 = p, d, C_U, A_0$.

   She then forms and sends to $\mathcal{T}$:

   $M_0 = p, ID_U, PKE_{PK_T}(K_0), E_{K_0}(X_0, SIGN_{SK_U}(X_0))$,

   where p is the protocol step identifier.

2. $\mathcal{U}$ forms the first log entry, L_0, with W_0 = **LogfileInitializationType** and $D_0 = d, d^+, ID_{log}, M_0$. Note that $\mathcal{U}$ does not store A_0 in the clear, as this could lead to replay attacks: an attacker gets control of $\mathcal{U}$ and forces a new audit log with the same A_0. $\mathcal{U}$ also stores $hash(X_0)$ locally while waiting for the response message.

3. $\mathcal{T}$ receives and verifies the initialization message. If it is correct (i.e., it decrypts correctly, $\mathcal{U}$'s signature is valid, $\mathcal{U}$ has a valid certificate), then $\mathcal{T}$ forms:

   $X_1 = p, ID_{log}, hash(X_0)$

   $\mathcal{T}$ then generates a random session key, K_1, and forms and sends:

   $M_1 = p, ID_T, PKE_{PK_U}(K_1), E_{K_1}(X_1, SIGN_{SK_T}(X_1))$.

4. $\mathcal{U}$ receives and verifies M_1. If all is well, then $\mathcal{U}$ forms a new log entry, L_j, with W_j=**ResponseMessageType** and $D_j = M_1$. $\mathcal{U}$ also calculates $A_1 = hash(\text{"Increment Hash"}, A_0)$.

   If $\mathcal{U}$ doesn't receive M_1 by the timeout time d^+, or if M_1 is invalid, then $\mathcal{U}$ forms a new log entry with W_j = **AbnormalCloseType** and D_j = the current timestamp and the reason for closure. The log file is then closed.

Depending on the application, we may or may not allow $\mathcal{U}$ to log anything between the time it sends M_0 and the time it receives M_1. In high-security applications, we might not want to take the chance that there are any problems with $\mathcal{T}$ or the communications. In other applications, it might be allowable for $\mathcal{U}$ to perform some actions while waiting for $\mathcal{T}$ to respond.

The purpose of writing the abort-startup message is simply to prevent there ever being a case where, due to a communications failure, $\mathcal{T}$ thinks there is a logfile being used even though none exists. Without this protection, an attacker could delete $\mathcal{U}$'s whole logfile after compromise, and claim to simply have failed to receive M_1 during the startup. In implementations where $\mathcal{U}$ waits for the response message before writing any log entries, M_1 will be the second message written in the log as well as the last. Otherwise, when $\mathcal{T}$ sees this message, he will believe either that $\mathcal{U}$ didn't receive the response message, or that $\mathcal{U}$ was compromised before the response message.

3.3 Closing the Logfile

Closing the logfile involves three operations: Writing the final-record message, D_f, (the entry code is **NormalCloseMessage** and the data is a timestamp), irretrieveably deleting A_f and K_f, and (in some implementations) actually closing the file. Note that after the file has been properly closed, an attacker who has taken control of $\mathcal{U}$ cannot make any alteration to the logfile without detection. Nor can an attacker delete some entries (and possibly add others) and then create a valid close-file entry earlier in the log. Finally, the attacker cannot delete the whole log file, because of the earlier interaction between $\mathcal{U}$ and $\mathcal{T}$. Any of these changes will be detected when $\mathcal{T}$ sees the final logfile.

3.4 Validating the Log

When $\mathcal{T}$ receives the complete and closed log, he can validate it using the hash chain and Z_f (since it already knows D_0). He can also derive all the encryption keys used, and thus read the whole audit log.

3.5 Verification, Verifiers, and Querying Entries

At times, a moderately-trusted person or machine, called $\mathcal{V}$, may need to verify or read some of the logfile's records while they are still on $\mathcal{U}$. This is made possible if $\mathcal{T}$ has sent M_1 to $\mathcal{U}$ (see Section 3.3), and if $\mathcal{V}$ has a high-bandwidth channel available to and from $\mathcal{U}$. Note that this can occur before $\mathcal{T}$ has received a copy of the log from $\mathcal{U}$, and before $\mathcal{U}$ has closed the logfile.

1. $\mathcal{V}$ receives a copy of the audit log, $L_0, L_1, L_2, \ldots, L_f$, where f is the index value of the last record, from $\mathcal{U}$. Note that $\mathcal{U}$ does not have to send $\mathcal{V}$ a complete copy of the audit log, but it must send $\mathcal{V}$ all entries from L_0 through some L_f, including the entry with M_1.

2. $\mathcal{V}$ goes through the hash chain in the log entries (the Y_i values), verifying that each entry in the hash chain is correct.

3. $\mathcal{V}$ establishes a secure connection with $\mathcal{T}$.

4. $\mathcal{V}$ generates a list of all the log entries she wishes to read, from 0 to n. This list contains a representation of the log entry type and index of each entry to which the verifier is requesting access. (Typically, only some log entry types will be allowed, in accordance with $\mathcal{V}$'s permission mask.) This list is called $Q[0..n]$, where $Q_i = j, W_j$.

5. $\mathcal{V}$ forms and sends to $\mathcal{T}$:

$$M_2 = p, ID_{log}, f, Y_f, Z_f, Q[0..n].$$

6. $\mathcal{T}$ verifies that the log has been properly created on $\mathcal{U}$, and that $\mathcal{V}$ is authorized to work with log. $\mathcal{T}$ knows A_0 so he can calculate A_f; this allows him to verifies that $Z_f = MAC_{A_f}(Y_f))$. If there is a problem, $\mathcal{T}$ sends the proper error code to $\mathcal{V}$ and records that there is a problem with ID_{log} on $\mathcal{U}$ or a problem with $\mathcal{V}$.

7. If there are no problems, $\mathcal{T}$ forms a list, $R[1..n]$, of responses to the requests in Q. Each entry in Q gets a responding entry in R: either giving the verifier the decryption key for that record, or else giving it an error code describing the reason for the refusal of access. (Typically, this will be because the log entry type isn't allowed to be read by this verifier.) Note that $\mathcal{T}$ computes these keys based on the log entry type codes given. If $\mathcal{V}$ provides incorrect codes to $\mathcal{T}$, the keys will be incorrect and $\mathcal{V}$ will be unable to decrypt the log entries. Additionally, $\mathcal{V}$ will not be able to derive the right key from the key he has been given.

8. $\mathcal{T}$ forms and sends to $\mathcal{V}$:

$$M_3 = p, R[0..n].$$

9. $\mathcal{V}$ is now able to decrypt and read, but not to change, the log entries whose keys were sent in M_4. $\mathcal{V}$ is also convinced that the log entries are authentic, since a correct MAC on any hash-chain value is essentially a MAC on all previous entries as well.

10. $\mathcal{V}$ deletes the key it established with $\mathcal{T}$ in Step (3). This guarantees that an attacker cannot read $\mathcal{U}$'s logfile if $\mathcal{V}$ is compromised later.

Of course, if $\mathcal{V}$ is compromised at the start of this protocol with $\mathcal{T}$, it will be able to read $\mathcal{U}$'s logfile. Presumably, $\mathcal{V}$ will have its own authenticated logfiles and will be regularly audited to reduce the likelihood of this problem occuring.

4 Applications and Extensions

An Offline Variant In the protocols and message formats given in Section 3.3, we left the specifics of the timing of M_1 from $\mathcal{T}$ to $\mathcal{U}$ open. This allows us to create a completely offline variant using couriers as the only communications medium. Thus, M_0 and M_1 can be sent via courier on diskettes. If a voice channel is available, someone can also read the hash of M_0 from $\mathcal{U}$ over the line for additional security.

Voice Line Only It is also possible to reduce the protocol in Section 3.3 to messages that can be sent over a voice line directly, either by reading them over the line or by using DTMF tones. In this case, $ID_U, ID_{log}, hash(M_0)$ must be read over the line. In practice, this can probably be reduced down to 22 digits (with the SHA-1 hash reduced to only 20 digits).[4] This will provide resistance against practical attacks that do not involve compromise of $\mathcal{U}$ before the logfile is created.

[4] We get moderate resistance to targeted collision attacks, but not to free collision atacks, with messages of 20 decimal digits.

4.1 Cross-Peer Linkages: Building a Hash Lattice

If there are multiple instances of $\mathcal{U}$ executing this same protocol, they can cross-link their audit logs with each other. In applications where there are many instances of $\mathcal{T}$ and with different instances of $\mathcal{U}$ authenticating their log files with different instances of $\mathcal{T}$, this cross-linking can make back-alteration to audit logs impractical, even with one or more compromised instances of $\mathcal{U}$ or even (in some cases) $\mathcal{T}$. This will also decrease the freedom of any compromised $\mathcal{U}$ machine to alter logs, because it keeps having to commit to its log's current state on other uncompromised machines.

This cross-authentication is done in addition to the rest of the scheme as described above. To cross-authenticate between two untrusted machines, $\mathcal{U}_0$ and $\mathcal{U}_1$, we execute the following protocol.

1. $\mathcal{U}_0$ and $\mathcal{U}_1$ exchange identities and establish a secure connection.

2. $\mathcal{U}_0$ forms and enters into its own log an entry, in position j, with:

   W_j = **"Cross Authentication Send"**,

   D_j = "Cross Authentication Send", $ID_{\mathcal{U}_1}, d_0$,

   where d_0 is $\mathcal{U}_0$'s timestamp.

3. $\mathcal{U}_0$ forms and sends to $\mathcal{U}_1$:

   $M_4 = p, Y_j, d_0.$

   Recall that Y_j is the current value of $\mathcal{U}_0$'s hash chain.

4. $\mathcal{U}_1$ receives this message, and verifies that the timestamp is approximately correct. If so, $\mathcal{U}_1$ writes a log entry in position i with:

   W_i = **"Cross Authentication Receive"**,

   D_i = "Cross Authentication Receive", $ID_{\mathcal{U}_0}$, d_0, Y_j.

   Then $\mathcal{U}_1$ forms and sends to $ID_{\mathcal{U}_0}$

   $M_5 = p, Y_i.$

   If $\mathcal{U}_1$ doesn't agree with the timestamp, it writes a log entry in position i with:

   W_i = **"Cross Authentication Receive Error**,

   D_i = "Cross Authentication Receive Error", $ID_{\mathcal{U}_0}$, d_0, d_1, Y_j,

   where d_1 is $\mathcal{U}_1$'s timestamp. Then $\mathcal{U}_1$ forms and sends to $\mathcal{U}_0$:

   $M_6 = p, Y_i, ErrorCode.$

   The protocol is then terminated.

5. $\mathcal{U}_0$ receives M_6 and processes it. If there was no error in receiving M_6, and if M_6 was not an error message, then $\mathcal{U}_0$ writes an entry to the log at position $j+1$ with:

   W_{j+1} = **"Cross Authentication Reply"**,

   D_{j+1} = "Cross Authentication Reply", $ID_{\mathcal{U}_1}$, Y_i.

   If it was an error, or if the expected message doesn't arrive before timeout, then $\mathcal{U}_0$ writes an entry to the log at position $j+1$ with:

   W_{j+1} = **"Cross Authentication Reply Error"**,

   D_{j+1} = "Cross Authentication Reply Error", $ID_{\mathcal{U}_1}$, ErrorCode.

If mutual cross-peer linking is required, $\mathcal{U}_1$ could then initiate this same protocol with $\mathcal{U}_0$.

This protocol could be useful in a network of electronic wallets. The wallets, $\mathcal{U}$s, would exchange money with each other regularly and occasionally come in contact with a banking terminal $\mathcal{T}$. This hash lattice could help the bank reconstruct fradulant transactions and trace wallets whose tamper-resistance has been defeated.

4.2 Replacing $\mathcal{T}$ with a Network of Insecure Peers

We can run this whole scheme using multiple untrusted machines, $\mathcal{U}_0, \mathcal{U}_1, ..., \mathcal{U}_n$, to do all the tasks of $\mathcal{T}$. Since $\mathcal{T}$ is a huge target, this could potentially increase security. Basically, this involves an extension of the hash lattice ideas from the previous section.

Let $\mathcal{U}_0$ and $\mathcal{U}_1$ both be untrusted machines, with $\mathcal{U}_0$ about to start an audit log. $\mathcal{U}_1$ will serve as the trusted server for this audit log.

1. $\mathcal{U}_0$ and $\mathcal{U}_1$ establish a secure connection.

2. $\mathcal{U}_0$ forms:

   d, a current timestamp.

   d^+, timestamp at which $\mathcal{U}_0$ will time out.

   ID_{log}, a unique identifier for this logfile.

   A_0, a random starting point.

   $ID_{\mathcal{U}_0}, ID_{\mathcal{U}_1}$ are unique identifiers for U_0 and U_1, respectively.

   $X_0 = p, ID_{\mathcal{U}_0}, ID_{\mathcal{U}_1}, d, ID_{log}, A_0$.

   She then forms and sends to $\mathcal{U}_1$

   $M_0 = X_0$.

3. $\mathcal{U}_0$ forms the first log entry with:

   $W_0 =$ **LogfileInitializationType**,

   $D_0 = d, d^+, ID_{log}, M_0$.

   Again, $\mathcal{U}_0$ does not store A_0 in the clear to protect against a replay attack. $\mathcal{U}_0$ also calculates and stores $hash(X_0)$ locally while waiting for the response message from $\mathcal{U}_1$.

4. $\mathcal{U}_1$ receives and verifies that M_0 is well formed. If it is, then $\mathcal{U}_1$ forms:

   $X_1 = p, ID_{log}, hash(X_0)$

   $\mathcal{U}_1$ then forms and sends to $\mathcal{U}_0$:

   $M_1 = X_1$.

5. $\mathcal{U}_0$ receives and verifies M_1. If all is well, then $\mathcal{U}_0$ forms a new log entry with:

   $W_0 =$ **ResponseMessageType**,

   $D_j = M_1$.

   If $\mathcal{U}_0$ doesn't receive M_1 by the timeout time d^+, or if M_1 is invalid, then $\mathcal{U}_0$ forms a new log entry with:

   $W_0 =$ **AbnormalCloseType**,

   $D_j =$ the current timestamp and the reason for closure.

   The log file is then closed.

One potential issue here is that an attacker may compromise $\mathcal{U}_1$, allowing the wholesale alteration of entries in the logfile. There are two imperfect solutions to this:

> $\mathcal{U}_0$ should log the same data in several parallel logfiles, with each logfile using a different untrusted server as its trusted server.

> $\mathcal{U}_0$ may commit, in the first protocol message, to a number N that will be the index number of its final log entry. $\mathcal{U}_1$ then computes A_N and $K_{0..N}$, stores these values, and deletes A_0. If an attacker compromises $\mathcal{U}_1$, he will learn what he needs to read and to close the logfile on $\mathcal{U}_0$, but not what he needs to alter any other log entries.

It is worth noting that these measures do not protect the secrecy of a logfile once an attacker has compromised both $\mathcal{U}_0$ and $\mathcal{U}_1$. An improved solution is for $\mathcal{U}_0$ to use a secret-sharing scheme to store A_0 among n untrusted machines, any m of which could then recover it.

Another solution is for $\mathcal{U}_0$ to keep parallel log files in the manner described above on n machines, but generating (for each log entry that needed to be kept secret) $n-1$ random bit strings of length equal to that of D_j. $\mathcal{U}_0$ then stores $D_j \oplus Pad_0 \oplus Pad_1 \oplus ... \oplus Pad_{N-2}$ in one logfile, and each pad value in another logfile.

In practice, these kinds of distributed schemes seem to work better for authenticating the log entries than for protecting their secrecy.

5 Summary and Conclusions

Many security systems, whether they protect privacy, secure electronic-commerce transactions, or use cryptography for something else, do not directly prevent fraud. Rather, they detect attempts at fraud after the fact, provide evidence of that fraud in order to convict the guilty in a court of law, and assume that the legal system will provide a "back channel" to deter further attempts. We believe that fielded systems should recognize this fundamental need for detection mechanisms, and provide audit capabilities that can survive both successful and unsuccessful attacks. Additionally, an unalterable log should make it difficult for attackers to cover their tracks, meaning that the victims of the attack can quickly learn that their machine has been attacked, and take measures to contain the damage from that attack. The victims could then revoke some public key certificates, inform users that their data may have been compromised, wipe the machine's storage devices and restoring it from a clean backup, or improve physical and network security on the machine to prevent further attacks.

In this paper, we have presented a general scheme that allows keeping an audit log on an insecure machine, so that log entries are protected even if the machine is compromised. We have shown several variations to this scheme, including one that is suitable for multiple electronic wallets interacting with each other but not connected to a central secure network. This scheme, combined with physical tamper-resistance and periodic inspection of the insecure machines, could form the basis for highly trusted auditing capabilities.

Our technique is strictly more powerful than simply periodically submitting audit logs and log entries to a trusted time-stamping server [HS91]: the per-record encryption keys and permission masks permit selective disclosure of log information, and there is some protection against dential-of-service attacks against the communications link between the insecure machine and the trusted server.

The primary limitation of this work is that an attacker can sieze control of an insecure machine and simply continue creating log entries, without trying to delete or change any previous log entries. In any real system, we envision log entries for things like "Tamper-resistance breach attmpt" that any successful attacker will want to remove. Even so, the possibility of an unlogged successful attack make it impossible to be certain that a machine was uncompromised before a given time. A sufficiently sneaky attacker might even create log entries for a phony attack hours after the real, unlogged, compromise.

In future work, we intend to expand this scheme to deal with the multiparty situation more cleanly. For example, we might like to be able to specify any three of some group of five nodes to play the part of the trusted machine. While this is clearly possible, we have not yet worked out the specific protocols. We also might to use ideas from the Rampart system [Rei96] to facilitate distributed trust. Also, it would be useful to anonymize the communications and protocols between an untrusted machine and several of its peers, which are playing the part of the trusted machine in our scheme. This would make it harder for an attacker to compromise one machine, and then learn from it which other machines to compromise in order to be able to violate the log's security on the first compromised machine.

6 Acknowledgments

The authors would like to thank Hilarie Orman and David Wagner, as well as the anonymous USENIX Security Sumposium referees, for their helpful comments; and David Wagner, again, for the LaTeX figure. This work is patent pending in the United States and other countries.

References

[And95] R. Anderson, "Robustness Principles for Public Key Protocols," *Advances in Cryptology — CRYPTO '95,* Springer-Verlag, 1995, pp. 236–247.

[BCK96] M. Bellare, R. Canetti, and H. Krawcyzk, "Keying Hash Functions for Message Authentication," *Advances in Cryptology — CRYPTO '96,* Springer-Verlag, 1996, pp. 1–15.

[DBP96] H. Dobbertin, A. Bosselaers, and B. Preneel, "RIPEMD-160: A Strengthened Version of RIEPMD," *Fast Software Encryption, Third International Workshop,* Springer-Verlag, 1996, pp. 71–82.

[ElG85] T. ElGamal, "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms," *IEEE Transactions on Information Theory,* V. IT-31, n. 4, 1985, pp. 469–472.

[HS91] S. Haber and W.S. Stornetta, "How to Time Stamp a Digital Document," *Advances in Cryptology — Crypto '90,* Springer-Verlag, 1991, pp. 437–455.

[KSW97] J. Kelsey, B. Schneier, and D. Wagner, "Protocol Interactions and the Chosen Protocol Attack," *1997 Protocols Workshop*, Springer-Verlag, 1997, to appear.

[KSH96] J. Kelsey, B. Schneier, and C. Hall, "An Authenticated Camera," *Twelfth Annual Computer Security Applications Conference,* IEEE Computer Society Press, 1996, pp. 24–30.

[LMM91] X. Lai, J. Massey, and S. Murphy, "Markov Ciphers and Differential Cryptanalysis," *Advances in Cryptology — CRYPTO '91*, Springer-Verlag, 1991, pp. 17–38.

[MOV97] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1997.

[NBS77] National Bureau of Standards, NBS FIPS PUB 46, "Data Encryption Standard," National Bureau of Standards, U.S. Department of Commerce, Jan 1977.

[NIST93] National Institute of Standards and Technology, NIST FIPS PUB 180, "Secure Hash Standard," U.S. Department of Commerce, May 1993.

[NIST94] National Institute of Standards and Technologies, NIST FIPS PUB 186, "Digital Signature Standard," U.S. Department of Commerce, May 1994.

[Rei96] M. K. Reiter, "Distributing trust with the Rampart toolkit," *Communications of the ACM*, v. 39, n. 4, Apr 1996, pp. 71–74.

[RSA78] R. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Communications of the ACM*, v. 21, n. 2, Feb 1978, pp. 120–126.

[Sch94] B. Schneier, "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)," *Fast Software Encryption, Cambridge Security Workshop Proceedings*, Springer-Verlag, 1994, pp. 191-204.

[Sch96] B. Schneier, *Applied Cryptography, Second Edition*, John Wiley & Sons, 1996.

[SK97] B. Schneier and J. Kelsey, "Automatic Event-Stream Notorization Using Digital Signatures," *Security Protocols, International Workshop, Cambridge, United Kingdom, April 1996 Proceedings*, Springer-Verlag, 1997, pp. 155–169.

[Sto89] C. Stoll, *The Cuckoo's Egg*, Doubleday, New York, 1989.

[Sti95] D.R. Stinson, *Cryptography: Theory and Practice*, CRC Press, 1995.

StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks*

Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton,† Jonathan Walpole,
Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
immunix-request@cse.ogi.edu, http://cse.ogi.edu/DISC/projects/immunix

Abstract

This paper presents a systematic solution to the persistent problem of buffer overflow attacks. Buffer overflow attacks gained notoriety in 1988 as part of the Morris Worm incident on the Internet. While it is fairly simple to fix individual buffer overflow vulnerabilities, buffer overflow attacks continue to this day. Hundreds of attacks have been discovered, and while most of the obvious vulnerabilities have now been patched, more sophisticated buffer overflow attacks continue to emerge.

We describe StackGuard: a simple compiler technique that virtually eliminates buffer overflow vulnerabilities with only modest performance penalties. Privileged programs that are recompiled with the StackGuard compiler extension no longer yield control to the attacker, but rather enter a fail-safe state. These programs require *no* source code changes at all, and are binary-compatible with existing operating systems and libraries. We describe the compiler technique (a simple patch to `gcc`), as well as a set of variations on the technique that trade-off between penetration resistance and performance. We present experimental results of both the penetration resistance and the performance impact of this technique.

1 Introduction

This paper presents a systematic solution to the persistent problem of buffer overflow attacks. Buffer overflow attack gained notoriety in 1988 as part of the Morris Worm incident on the Internet [23]. Despite the fact that fixing individual buffer overflow vulnerabilities is fairly simple, buffer overflow attacks continue to this day, as reported in the SANS Network Security Digest:

> Buffer overflows appear to be the most common problems reported in May, with degradation-of-service problems a distant second. Many of the buffer overflow problems are probably the result of careless programming, and could have been found and corrected by the vendors, before releasing the software, if the vendors had performed elementary testing or code reviews along the way.[4]

The base problem is that, while individual buffer overflow vulnerabilities are simple to patch, the vulnerabilities are profligate. Thousands of lines of legacy code are still running as privileged daemons (`SUID root`) that contain numerous software errors. New programs are being developed with more care, but are often still developed using unsafe languages such as C, where simple errors can leave serious vulnerabilities.

The continued success of these attacks is also due to the "patchy" nature by which we protect against such attacks. The life cycle of a buffer overflow attack is simple: A (malicious) user finds the vulnerability in a highly priv-

*This research is partially supported by DARPA contracts F30602-96-1-0331 and F30602-96-1-0302.

†Ryerson Polytechnic University

ileged program and someone else implements a patch to *that particular attack, on that privileged program.* Fixes to buffer overflow attacks attempt to solve the problem at the source (the vulnerable program) instead of at the destination (the stack that is being overflowed).

This paper presents StackGuard, a systematic solution to the buffer overflow problem. StackGuard is a simple compiler extension that limits the amount of damage that a buffer overflow attack can inflict on a program. Programs compiled with StackGuard are safe from buffer overflow attack, regardless of the software engineering quality of the program.

Section 2 describes buffer overflow attacks in detail. Section 3 details how StackGuard defends against buffer overflow attacks. Section 4 presents performance and penetration testing of StackGuard-enhanced programs. Section 5 discusses some of the abstract ideas represented in StackGuard, and their implications. Section 6 describes related work in defending against buffer overflow attack. Finally, Section 7 presents our conclusions.

2 Buffer Overflow Attacks

Buffer overflow attacks exploit a lack of bounds checking on the size of input being stored in a buffer array. By writing data *past* the end of an allocated array, the attacker can make arbitrary changes to program state stored adjacent to the array. By far, the most common data structure to corrupt in this fashion is the stack, called a "stack smashing attack," which we briefly describe here, and is described at length elsewhere [15, 17, 21].

Many C programs have buffer overflow vulnerabilities, both because the C language lacks array bounds checking, and because the culture of C programmers encourages a performance-oriented style that avoids error checking where possible [14, 13]. For instance, many of the standard C library functions such as `gets` and `strcpy` do not do bounds checking by default.

The common form of buffer overflow exploitation is to attack buffers allocated on the stack. Stack smashing attacks strive to achieve two mutually dependent goals, illustrated in Figure 1:

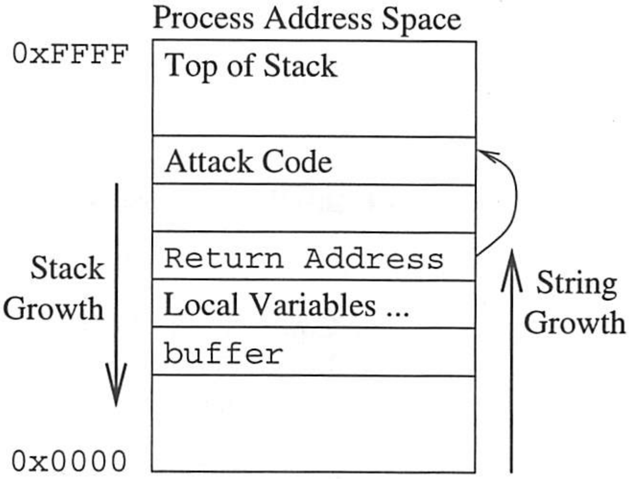


Figure 1: Stack Smashing Buffer Overflow Attack

Inject Attack Code The attacker provides an input string that is actually executable, binary code native to the machine being attacked. Typically this code is simple, and does something similar to `exec("sh")` to produce a `root` shell.

Change the Return Address There is a stack frame for a currently active function above the buffer being attacked on the stack. The buffer overflow changes the return address to point to the attack code. When the function returns, instead of jumping back to where it was called from, it jumps to the attack code.

The programs that are attacked using this technique are usually privileged daemons; programs that run under the user-ID of `root` to perform some service. The injected attack code is usually a short sequence of instructions that spawns a shell, also under the user-ID of `root`. The effect is to give the attacker a shell with `root`'s privileges.

If the input to the program is provided from a locally running process, then this class of vulnerability may allow any user with a local account to become `root`. More distressing, if the program input comes from a network connection, this class of vulnerability may allow any user anywhere on the network the ability to become root on the local host. Thus while new instances of this class of attack are not intellectually interesting, they are none the less critical to practical system security.

Engineering such an attack from scratch is non-trivial. Often, the attacks are based on reverse-engineering the attacked program, so as to determine the exact offset from the buffer to the return address in the stack frame, and the offset from the return address to the injected attack code. However, it is possible to soften these exacting requirements [17]:

- The location of the return address can be approximated by simply repeating the desired return address several times in the approximate region of the return address.
- The offset to the attack code can be approximated by prepending the attack code with an arbitrary number of NOP instructions. The overwritten return address need only jump into the middle of the field of NOPs to hit the target.

The cook-book descriptions of stack smashing attacks [15, 17, 21] have made construction of buffer-overflow exploits quite easy. The only remaining work for a would-be attacker to do is to find a poorly protected buffer in a privileged program, and construct an exploit. Hundreds of such exploits have been reported in recent years [4].

3 StackGuard: Making the Stack Safe for Network Access

StackGuard is a compiler extension that enhances the executable code produced by the compiler so that it detects and thwarts buffer-overflow attacks against the stack. The effect is transparent to the normal function of programs. The only way to notice that a program is StackGuard-enhanced is to cause it to execute C statements with undefined behavior: StackGuard-enhanced programs define the behavior of writing to the return address of a function *while* it is still active.

As described in Section 2, the common form of buffer-overflow attacks are stack smashers. They function by overflowing a buffer that is allocated on the stack, injecting code onto the stack, and changing the return address to point to the injected code. StackGuard thwarts this class of attack by effectively preventing changes to the return address while the function is still active. If the return address cannot be changed, then the attacker has no way of invoking the injected attack code, and the attack method is thwarted.

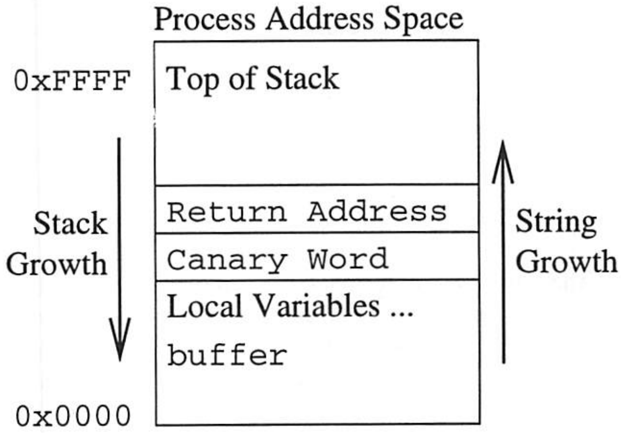


Figure 2: Canary Word Next to Return Address

StackGuard prevents changes to active return addresses by either detecting the change of the return address *before* the function returns, or by completely preventing the write to the return address. Detecting changes to the return address is a more efficient and portable technique, while preventing the change is more secure. StackGuard supports both techniques, as well as adaptively switching from one mode to the other.

Section 3.1 describes how StackGuard detects changes to the return address. Section 3.2 describes how StackGuard prevents changes to the return address. Section 3.3 discusses motives and methods for adaptively switching between techniques.

3.1 Detecting Return Address Change Before Return

To be effective, detecting that the return address has been altered must happen *before* a function returns. StackGuard does this by placing a "canary"[1] word next

[1] A direct descendent of the Welsh miner's canary.

to the return address on the stack, as shown in Figure 2. When the function returns, it first checks to see that the canary word is intact before jumping to the address pointed to by the return address word.

This approach assumes that the the return address is unaltered *IFF* the canary word is unaltered. While this assumption is not completely true in general (stray pointers can alter any word), it *is* true of buffer overflow attacks. The buffer overflow attack method exploits the fact that the return address word is located very close to a byte array with weak bounds checking, so the *only* tool the attacker has is a linear, sequential write of bytes to memory, usually in ascending order. Under these restricted circumstances, it is very difficult to over-write the return address word without disturbing the canary word.

The StackGuard implementation is a *very* simple patch to `gcc` 2.7.2.2. The `gcc` `function_prologue` and `function_epilogue` functions have been altered to emit code to place and check canary words. The changes are architecture-specific (in our case, `i386`), but since the total changes are under 100 lines of `gcc`, portability is not a major concern. All the changes in the gcc calling conventions are undertaken by the callee, so code compiled with the StackGuard-enhanced `gcc` is completely inter-operable with generic `gcc .o` files and libraries. The additional instructions added to the function prologue are shown in pseudo-assembly form in Figure 3, and the additional instructions added to the instruction epilogue are shown in Figure 4. Section 4 describes testing and performance of this patch.

3.1.1 Randomizing the Canary

The Canary defense is sufficient to stop most buffer overflow attacks that are oblivious to the canary. In fact, simply *changing* the compiler's calling conventions is sufficient to stop most buffer overflow attacks [8]. Most *current* buffer overflow attacks are quite brittle, making specific, static assumptions about the layout of the stack frame. However, it is not very hard for attackers to develop buffer overflows that are insensitive to minor changes in the stack frame layout [17]:

- To adapt to changes in the location of the return address relative to the buffer being overflowed, the attacker can repeat the new value several times in the input string.
- To adapt to imprecision in the offset of the injected code from the current program counter, the attacker can inject attack code consisting of many `NOP`s, and simply jump to somewhere in the middle of the `NOP` sequence. Control flow will then drop down to the attack code.
- To adapt to changes in alignment, the attacker need only guess 4 times at most to get the alignment correct.

It is also possible to write attacks *specifically* designed to overcome StackGuard.[2] There are two ways to overcome the Canary method of detecting buffer overflows:

1. Skip over the canary word. If the attacker can locate a poorly checked copy of an array of structs, which have alignment requirements, and are not big enough to fulfill the alignment requirements while densely packing the array, then it is possible that the copy could occur such that the canary word is in one of the holes left in the array. We expect this form of vulnerability to be rare, and difficult to exploit.
2. Simulate the canary word. If the attacker can easily guess the canary value, then the attack string can include the canary word in the correct place, and the check at the end of the function. If the canary word is completely static, then it is *very* easy to guess. This form of attack is problematic.

To deal with easily-guessed canaries, we use randomly chosen canary values. Our current implementation enhances the `crt0` library to choose a set of random canary words at the time the program starts. These random words are then used as distinct random canary words, one per function in the object code. While it is not *impossible* to guess such a canary value, it is difficult: the attacker must be able to examine the memory image of the running process to get the randomly selected word. Even so, a determined attacker could break such a defense eventually; we discuss our approach to this problem in Section 3.3.

[2] Naturally, none have been found to date :-)

```
    move canary-index-constant into register[5]
    push canary-vector[register[5]]
```

Figure 3: Function Prologue Code: Laying Down a Canary

```
    move canary-index-constant into register[4]
    move canary-vector[register[4]] into register[4]
    exclusive-or register[4] with top-of-stack
    jump-if-not-zero to constant address .canary-death-handler
    add 4 to stack-pointer
    < normal return instructions here>
.canary-death-handler:
    ...
```

Figure 4: Function Epilogue Code: Checking a Canary

3.2 Preventing Return Address Changes With MemGuard

The Synthetix project [18, 1, 2, 24] introduced a notion called "*quasi*-invariants." Quasi-invariants are state properties that hold true for a while, but may change without notice. Quasi-invariants are used to specify *optimistic* specializations: code optimizations that are valid *only* while the quasi-invariants hold. We have extended this work to treat return addresses on the stack as quasi-invariant during the activation lifetime of the function. The return address is read-only (invariant) while the function is active, thus preventing effective buffer overflow against the stack.

MemGuard [3] is a tool developed to help debug optimistic specializations by locating code statements that change quasi-invariant values. MemGuard provides fine-grained memory protection: individual words of memory (quasi-invariant terms) can be designated as read-only, except when explicitly written via the MemGuard API. We have used MemGuard to produce a more secure, if less performant, version of the StackGuard compiler.

```
push a
push b
move 164 into a
move arg[0] into b
trap 0x80
pop b
pop a
```

Figure 5: Function Prologue Code: Protecting the Return Address With MemGuard

MemGuard is used to prevent buffer overflow attacks by protecting a return address when a function is called, and un-protecting the return address when the function returns. The protection and un-protection occur in precisely the same places as the canary placement and checks described in Section 3.1: the `function_prologue` and `function_epilogue` functions. Figure 5 shows the prologue code sequence for MemGuard. The epilogue code sequence is identical, but uses system call 165 instead of 164.

MemGuard is implemented by marking virtual memory pages containing quasi-invariant terms as read-only, and installing a trap handler that catches writes to protected pages, and emulates the writes to non-protected words on protected pages. The cost of a write to a non-protected word on a protected page is approximately 1800 times the cost of an ordinary write. This is an acceptable cost when quasi-invariant terms are in quiet portions of the kernel's address space, and when MemGuard is primarily used for debugging.

This cost is not acceptable when the protected words are located near the top of the stack, next to some of the most frequently written words in the program. MemGuard was originally designed to protect variables within the kernel. To protect the stack, MemGuard had to be extended in several ways:

- Extend VM model to protect user pages.
- Deal with the performance penalties due to "false sharing" caused by frequent writes to words near the return address.
- Provide a light-weight system-call interface to MemGuard. Loading virtual memory hardware is a privileged operation, and so the application process must trap to kernel mode to protect a word.

Most of these extensions are simple software development, but the performance problems are challenging. Fortunately, the Pentium processor has four "debug" registers. These registers can be configured to watch for read, write, and execute access to the virtual address loaded into each register, and generate an exception when such access occurs.

We use these registers as a cache of the most recently protected return addresses. The goal is to eliminate the need for the top-most page of the stack to be read-only, to eliminate page faults resulting from writes to variables at the top of the stack. Because of the locality behavior of stack variables, restoring write privileges to the top of the stack should handle most of the writes to stack variables.

It is only probabilistically true that protecting the four most recent return addresses will capture all protection needs for the top of the stack. However, if the compiler is adjusted to emit stack frames with a minimum size of 1/4 of a page, then it is always true that 4 registers will cover the top page. The time/space trade-off implied by this approach can be continuously adjusted, reducing the minimum size of stack frames to reduce space consumption, and also increasing the probability that the top page of the stack actually will require MemGuard protection, with its associated costs.

3.3 Adaptive Defense Strategies

StackGuard is a product of the Immunix project [11], whose focus is adaptive responses to security threats. Thus we provide an adaptive response to intrusions, switching between the more performant Canary version, and the more robust MemGuard versions of StackGuard.

The basic model of operation for StackGuard is that when a buffer overflow is detected, either by the Canary or by MemGuard, the process is terminated. The process must exit, because an unknown amount of state has already been corrupted at the time the attack is detected, and so it is impossible to safely recover the state of the process. Thus the process exits, using only static data and code, so as to avoid any possible corruption from the attacker.

Replacing the dead process is context-dependent. In many cases, it suffices to just let `inetd` re-start the daemon when a connection requests service. However, if the daemon is not managed by `inetd`, then it may be necessary for a watch-dog process to re-start the daemon, most especially in the case of `inetd` itself.

It is also possible for these re-start mechanisms to adaptively select which form of protection to use *next*. The Canary and MemGuard variants of StackGuard offer different points in the trade-off between security and performance. The Canary version is more performant, while the MemGuard version is more secure (see Section 4). More specifically, the important security vulnerability in the Canary variant is that it is potentially subject to guessing of the canary value. The Canary variant can defend itself against guessing by exiting, and replacing the attacked Canary-guarded daemon with a MemGuard-guarded daemon.

This adaptive response allows systems to run in a relatively high-performance state most of the time, and adaptively switch to a lower-performance, higher-security state when under attack. At worst, the attacker can carry out a degradation-of-service attack by periodically attacking daemons, forcing them to run in the lower-performance MemGuard mode most of the time. However, service is not totally denied, because the daemons continue to function, and the attacker no longer is able to obtain illegitimate privilege via buffer overflow attack.

4 Experimental Results

This section describes experimental evaluation of StackGuard. Subsection 4.1 describes penetration experiments, to show StackGuard's effectiveness in deterring past and future attacks. of Subsection 4.2 describes the performance cost of StackGuard under various circumstances.

4.1 StackGuard Effectiveness

Here we illustrate StackGuard's effectiveness in thwarting stack smashing buffer overflow attacks. StackGuard is intended to thwart generic stack smashing attacks, even those that have not yet appeared. To simulate that, we sought out buffer overflow exploits, and tried them against their intended software targets, with and without protection from StackGuard. Table 1 summarizes these results.

The programs listed in Table 1 are conventionally installed as `SUID root`. If the attacker can get one of these programs to start a shell, then the attacker gets a `root` shell.

In each case, the experiment is to install the vulnerable program `SUID root` (`SUID httpd` for `wwwcount`) and attack it with the exploit. We then re-compile the program with the Canary variant of StackGuard, re-install the StackGuard-enhanced program as `SUID root`, and attack it again with the exploit. We did *not* alter the source code of any of the vulnerable programs at all, and StackGuard has no specific knowledge of any of these attacks. Thus this experiment *simulates* the effect of StackGuard defending against unknown attacks.

In all cases we have studied, both the Canary and the MemGuard variants of StackGuard stopped what would have been an attack that obtains a `root` shell. Several cases deserve special discussion:

`umount 2.5k/libc 5.3.12`: The buffer overflow vulnerability is actually in `libc`, and not in `umount`. Simply re-compiling `umount` with either variant of StackGuard does not suffice to stop the attack. However, when `libc` is also compiled using StackGuard (either variant) then the attack is defeated. Thus for full protection, either the system shared libraries must be protected with StackGuard, or the privileged programs must be statically linked with libraries that are protected with StackGuard.

`SuperProbe`: This attack does not actually attack the function return address. Rather, it over-writes a function pointers in the program that is allocated on the stack. The Canary variant stopped the attack by perturbing the layout of the stack, but an adjusted attack produced a `root` shell even with Canary protection. The MemGuard variant stopped the attack because a return address was in the way of the buffer overflow. Proper treatment of this kind of attack requires an extension to StackGuard, as described in Section 5.4.

`Perl`: Like `SuperProbe`, the `Perl` attack does not attack the function return address. This attack overwrites data structures in the global data area, and thus is not properly a "stack smashing" attack. Permutations in the alignment of the global data area induced by the StackGuard's vector of canary values prevented the attack from working, but a modified form of the attack produced a `root` shell despite Canary protection. MemGuard had no effect on the attack.

`Samba`, `wwwcount`: These buffer overflow vulnerabilities were announced *after* the StackGuard compiler was developed, yet the StackGuard-enhanced versions of these programs were *not* vulnerable to the attacks. This illustrates the point that StackGuard can effectively prevent attacks even against unknown vulnerabilities.

We would like the list of programs studied to be larger. Two factors limit this kind of experimentation:

Obtaining the Exploit: It is difficult to obtain the exploit code for attacking programs. Security organizations such as CERT are reluctant to release exploits, and thus most of these exploits were obtained either from searching the web, or from the bugtraq mailing list [16].

Obtaining Vulnerable Source Code: Buffer overflow attacks exploit specific, *simple* vulnerabilities in popular software. Because of the severe security

Vulnerable Program	Result Without StackGuard	Result With Canary StackGuard	Result With MemGuard StackGuard
dip 3.3.7n	root shell	program halts	program halts
elm 2.4 PL25	root shell	program halts	program halts
Perl 5.003	root shell	program halts irregularly	root shell
Samba	root shell	program halts	program halts
SuperProbe	root shell	program halts irregularly	program halts
umount 2.5k/libc 5.3.12	root shell	program halts	program halts
wwwcount v2.3	httpd shell	program halts	program halts
zgv 2.7	root shell	program halts	program halts

Table 1: Protecting Vulnerable Programs with StackGuard

risks posed, and the ease of patching the individual vulnerability, new releases appear soon after the vulnerability is publicized. Moreover, the vulnerability is often *not* publicized until it can be announced with a patch in hand. The older vulnerable source code is often not easily available. We have begun archiving source code versions, so that we will be able to add experiments as new vulnerabilities appear.

4.2 StackGuard Overhead

This section describes experiments to study the performance overhead imposed by StackGuard. Note that StackGuard need only be used on programs that are SUID root, and such programs are not usually consumers of large amounts of CPU time. Thus it is only necessary that the overhead be sufficiently low that the privileged administrative daemons do not impose a noticeable compute load. The MemGuard and Canary variants of StackGuard impose different kinds of overhead, and so we microbenchmark them separately in Sections 4.2.1 and 4.2.2. Section 4.2.3 presents macrobenchmark performance data.

4.2.1 Canary StackGuard Overhead

The Canary mechanism imposes additional cost at two points in program execution:

- function prologue: there is a small cost in pushing the canary word onto the stack.
- function epilogue: there is a moderate cost in checking that the canary word is intact before performing the function return.

We model this cost as a % overhead per function call. The % overhead is a function of the base cost of a function call, which varies depending on the number of arguments and the return type, so we studied a range of function types.

The experiments seek to discover the % overhead of a function call imposed by StackGuard. We did this by writing a C program that increments a statically allocated integer 500,000,000 times. The base case is just "i++", and the experiments use various functions to increment the counter. The results are shown in Table 2. All experiments were performed on a 200 MHz Pentium-S with 512K of level 2 cache, and 128M of main memory.

The "i++" is the base case, and thus has no % overhead. The "void inc()" entry is a function that does i++ where i is a global variable; this shows the overhead of a zero-argument void function, and is the worst-possible case, showing a 125% overhead on function calls. The "void inc(int *)" entry is a function that takes an int * argument and increments it as a side-effect; this shows that there is 69% overhead on a one-argument void function. The "int inc(int)" entry is an applicative function that takes an int argument, and returns that value + 1; this shows that the overhead of a one-argument function returning an int is 80%.

Increment Method	Standard Run-Time	Canary Run-Time	% Overhead
`i++`	15.1	15.1	NA
`void inc()`	35.1	60.2	125%
`void inc(int *)`	47.7	70.2	69%
`int inc(int)`	40.1	60.2	80%

Table 2: Microbenchmark: Canary Function Call Overhead

Numerous other experiments are possible, but they all increase the base cost of function calls, while the cost of the Canary mechanism remains fixed at 7 instructions (see Figures 3 and 4), decreasing the Canary % overhead. Thus these overhead microbenchmarks can be considered an upper-bound on the cost of the Canary compiler.

4.2.2 MemGuard StackGuard Overhead

The MemGuard variant of StackGuard suffers substantial performance penalties compared to the Canary variant, for reasons described in Section 3.2. Section 4.1 showed that the MemGuard variant provides better security protection for stack attacks than the Canary variant (specifically, MemGuard stopped the `SuperProbe` attack, and guessing canary values will not help get past MemGuard). This section measures the cost of that added protection.

The MemGuard variant of StackGuard is still under development, but as of this writing, we have some preliminary results. We have measured the performance of two versions of MemGuard StackGuard:

MemGuard Register This version uses *only* the Pentium's debugging registers for protection, so only the four most recent function calls' return addresses are protected. This version pays no penalty for page protection faults induced by protecting the stack with virtual memory protection. **NOTE**: this version stopped **all** of the stack smashing attacks that we tested[3].

MemGuard VM This version uses the virtual memory page protection scheme described in Section 3.2. It has not fully exploited the optimization of using the debugging registers as a cache, to keep the top page of the stack writable. Thus this version suffers *substantial* performance penalties due to a large number of page protection faults.

Table 3 shows the overhead costs for the MemGuard variant of StackGuard. Because of the use of a heavyweight system call to access privileged hardware for protection, function calls slow down by 70× for the MemGuard Register protection. The additional penalty of page protection fault handling for false sharing of the page on the top of the stack raises the cost of function calls by 160×. Proper use of the debugging registers as a cache for the VM mechanism should bring the costs in line with the MemGuard Register costs.

4.2.3 StackGuard Macrobenchmarks

Sections 4.2.1 and 4.2.2 present microbenchmark results on the additional cost of function calls in programs protected by StackGuard. However, these measurements are *upper bounds* on the real costs of running programs under StackGuard; the true penalty of running StackGuard-enhanced programs is the overall cost, not the microbenchmark cost. We have benchmarked two programs: `ctags`, and the StackGuard-enhanced `gcc` compiler itself.

The `ctags` program constructs an index of C source code. It is 3000 lines of C source code, comprising 68 separate functions. When run over a small set of source files (78 files, 37,000 lines of code) with a hot buffer cache, `ctags` is completely compute-bound. When run over a large set of files (1163 files, 567,000 lines of code) `ctags` it is still compute-bound, because of the large

[3] Except `Perl`, which is not really a stack smashing attack.

Increment Method	Standard Run-Time	MemGuard Register Run-Time	% Overhead	MemGuard VM Run-Time	% Overhead
`i++`	15.1	15.1	NA	NA	NA
`void inc()`	35.1	1808	8800%	34,900	174,300%
`void inc(int *)`	47.7	1820	5400%	40,420	123,800%
`int inc(int)`	40.1	1815	7000%	41,610	166,200%

Table 3: Microbenchmark: MemGuard Function Call Overhead

amount of RAM in our test machine.

On a smaller machine, the test becomes I/O bound, consuming 50% of the CPU's time, so it is approximately balanced. While the Canary variant still consumes more CPU time than the generic program, it is overlapped with disk I/O, and the program completes in the same amount of real time. The MemGuard variants consume so much CPU time that the program's real time is dramatically impacted.

Table 4 shows ctag's run-time in these two cases. The Canary variant's performance penalties are moderate, at 80% for the small case, and 42% for the large case. The MemGuard Register penalties are substantial, at 1100% for the small case, and 1000% for the large case. The MemGuard VM performance penalties are prohibitive, at 46,000% for the small case, and 36,000% for the large case.

Table 5 shows a similar experiment for the run-time of a StackGuard-protected gcc compiler. We thus use a StackGuard-protected gcc to measure the performance cost of StackGuard for a large and complex program. To be clear, the experiment measures the cost of running gcc protected by StackGuard, and only incidentally measures the cost of adding StackGuard protection to the compiled program.

Table 5 shows the time to compile ctags using gcc enhanced with StackGuard. Because there is more computation per function call for gcc than ctags, this time the costs are lower. The Canary version consumes only 6% more CPU time, and only 7% more real time. The MemGuard variants benefited as well; the Register version's additional real time cost is 214%, and the VM version's additional cost is 5100%.

Recall that the StackGuard protective mechanism is only necessary on privileged administrative programs. Such programs present only a minor portion of the compute load on a system, and so the StackGuard overhead will have only a modest impact on the total system load. Thus the overhead measured here could be considered within reason for heightened security, without a significant change in the administrative complexity of the system. We discuss administration of StackGuard in Section 5.

5 Discussion

This section discusses some of the abstract ideas represented in StackGuard, and their implications. Section 5.1 describes how StackGuard can help defend against *future* attacks. Section 5.2 describes potential administration and configuration techniques for systems using StackGuard. Section 5.3 describes some possible performance optimizations. Section 5.4 describes future enhancements to StackGuard.

5.1 Defending Against Future Attacks

Fundamentally, the attacks that StackGuard prevents are not very interesting. They are serious security faults that result from minor programming errors. Once discovered, fixing each error is easy. The significant contribution that StackGuard makes is not only that it patches a broad collection of existing faults, but rather that it patches a broad collection of *future* faults that have yet to be discovered. That StackGuard defeats the attacks against Samba and wwwcount discovered *after* StackGuard was produced is testament to this effect.

Input	Version	User Time	System Time	Real Time
37,000 lines	Generic	0.41	0.14	0.55
	Canary	0.68	0.13	0.99
	MemGuard Register	1.30	5.45	6.84
	MemGuard VM	16.5	238.0	255.1
586,000 lines	Generic	7.74	2.08	10.2
	Canary	11.9	2.07	14.5
	MemGuard Register	21.1	91.5	115.0
	MemGuard VM	236	3482	3728

Table 4: Macrobenchmark: `ctags`

Version	User Time	System Time	Real Time
Generic	1.70	0.12	1.83
Canary	1.79	0.16	1.96
MemGuard Register	2.22	3.35	5.76
MemGuard VM	8.17	87.7	96.2

Table 5: Macrobenchmark: `gcc` of the `ctags` program

Using StackGuard does not eliminate the need to fix buffer overflow vulnerabilities, but by converting `root` vulnerabilities into mild degradation-of-service attacks, it does eliminate the urgency to fix them. This gives software developers the breathing room to fix buffer overflows when it is convenient (i.e. when the next release is ready) rather than having to rush to create and distribute a patch. More importantly, StackGuard eases security administration by relieving the system administrators of the need to apply these patches as soon as they are released, often several times a month.

5.2 Administration and Configuration

The adaptive response described in Section 3.3 requires management: StackGuard causes programs to give notice that they need to be replaced because they have been (unsuccessfully) attacked, but does not make policy about what version, if any, to replace it with.

Different policy decisions will have different implications; switching to a higher level of protection will drastically reduce performance, yet failure to switch can lead to successful penetration via guessing. The decision to revert to the more performant, less secure mode is even more difficult, because the attacker may try to *induce* such a switch. Making the right choice, automatically, is challenging. We propose to create a small, domain-specific language [19] for specifying these policy choices.

StackGuard comes with a performance price, and can be viewed as an insurance policy. If one is *very* sure that a program is correct, i.e. contains no buffer overflow vulnerabilities because it has been verified using formal methods, or a validation tool [9], then the program can be re-compiled and installed without benefit of StackGuard.

StackGuard offers powerful protection of any program compiled with the StackGuard compiler, but does nothing for programs that have not been thus compiled. However, tools such as COPS [7], which search for programs that should not be `SUID root`, can be configured to look for programs that are `SUID root`, and have not been compiled using StackGuard or some other security verification tool [9]. If COPS reports that all `SUID root` programs on a machine have been protected, then one can have some degree of assurance that the machine is not vulnerable to buffer overflow attacks.

5.3 Performance Optimizations

Section 4.2.2 mentions that a light-weight trap to kernel mode can reduce the overhead of the MemGuard mechanism. However, it is also possible for the compiler to optimize StackGuard performance, both for the MemGuard and Canary variants.

If it is the case that no statement takes the address of any stack variable in the function `foo`, then `foo` does not need StackGuard protection. This is because any buffer overflow must attack an array, which is always a pointer. If an attack seeks to alter a variable in a function above `foo` on the stack, then it must come from below `foo`. But to get to the variable above `foo` it would have to go through the StackGuard protection that necessarily exists on the function below `foo` because of the array being overflowed.

The information regarding whether any variable has been aliased is already available in `gcc`, so it should be a simple matter to turn StackGuard protection off for functions that do not need it. We are working on this optimization, and expect to have it available in a future release of StackGuard.

5.4 Future Work

StackGuard defends against stack smashing buffer overflow attacks that over-write the return address and inject attack code. While this is the most common form of buffer overflow attack, it is not the only form, as illustrated by `SuperProbe` in Section 4.1.

In the general case, buffer overflow attacks can write arbitrary data to arbitrary pieces of process state, with arbitrary results limited only by the opportunities offered by buggy programs. However, some data structures are far easier to exploit than others. Notably, function pointers are highly susceptible to buffer overflow attack. An attacker could conceivably use a buffer overflow to overwrite a function pointer that is on the heap, pointing it to attack code injected into some other buffer on the heap. The attack code need not even overflow its buffer.

We propose to treat this problem by extending StackGuard to protect other data sensitive structures in addition to function return addresses. "Sensitive data structures" would include function pointers, as well as other structures as indicated by the programmer, or clues in the source code itself.

This extension highlights a property of StackGuard, which is that it is "destination oriented." Rather than trying to prevent buffer overflow attacks at the source, StackGuard strives to defend that which the attacker wants to alter. Following the notion that a TCB should be small to be verifiable (and thus secure) we conjecture that the set of data structures needing defending is smaller than the set of data structures exposed to attackers. Thus it should be easier to defend critical data structures than to find all poorly defended interfaces.

6 Related Work

There have been several other efforts pertinent to the problem of buffer overflow attacks. Some are explicitly directed at the security problem, while others are more generally concerned with software correctness. This section reviews some of these projects, and compares them against StackGuard. The result is not a conclusion of which approach is better, but rather a description of the different trade-offs that each approach provides.

6.1 Non-Executable Stack

"Solar Designer" has developed a Linux patch that makes the stack non-executable [6], precisely to address the stack smashing problem. This patch simply makes the stack portion of a user process's virtual address space non-executable, so that attack code injected onto the stack cannot be executed. This patch offers the advantages of *zero* performance penalty, and that programs work and are protected without re-compilation. However, it does necessitate running a specially-patched kernel, unless this extension is adopted as standard.

This patch was non-trivial and non-obvious, for the following reasons:

- `gcc` uses executable stacks for function trampolines for nested functions.

- Linux uses executable user stacks for signal handling.

- Functional programming languages, and some other programs, rely on executable stacks for run-time code generation.

The patch addresses the problem of trampolines and other application use of executable stacks by detecting such usage, and permanently enabling an executable stack for that process. The patch deals with signal handlers by dynamically enabling an executable stack only for the duration of the signal handler. Both of these compromises offer potential opportunities for intrusion, e.g. a buffer overflow vulnerability in a signal handler.

In addition to the above vulnerabilities, making the stack non-executable fails to address the problem of buffer overflow attacks that do not place attack code on the stack. The attacker may inject the attack code into a heap-allocated or statically allocated buffer, and simply re-point a function return address or function pointer to point to the attack code. This is exactly the kind of attack brought against `Perl` as described in Section 4.1, and a non-executable stack is no more effective than the current StackGuard in stopping it.

The attacker may not even need to inject attack code at all, if the right code fragment can be found within the body of the program itself. Thus additional protection for critical data structures such as function pointers and function return addresses, as described in Section 5.4.

6.2 FreeBSD Stack Integrity Check

Alexander Snarskii developed a FreeBSD patch [22] that does similar integrity checks to those used by the Canary variant of StackGuard. However, these integrity checks were non-portable, hard-coded in assembler, and embedded in `libc`. This method protects against stack smashing attacks inside `libc`, but is not as general as StackGuard.

6.3 Array Bounds Checking for C

Richard Jones and Paul Kelly have developed a `gcc` patch [12] that does full array bounds checking for C programs. Programs compiled with this patch are compatible with ordinary `gcc` modules, because they have not changed the representation of pointers. Rather, they derive a "base" pointer from each pointer expression, and check the attributes of that pointer to determine whether the expression is within bounds.

The performance costs are substantial: a pointer-intensive program (ijk matrix multiply) experienced 30× slowdown. Since the slowdown is proportionate to pointer usage, which is quite common in privileged programs, this performance penalty is particularly unfortunate.

However, this method is strictly more secure than StackGuard, because it will prevent all buffer overflow attacks, not just those that attempt to alter return addresses, or other data structures that are perceived to be sensitive (see Section 5.4). Thus we propose that programs compiled with the bounds-checking compiler be treated as the "backing store" for MemGuard-protected programs, just as MemGuard-protected programs are the back-up plan for Canary-protected programs (see Section 3.3).

6.4 Memory Access Checking

Purify [10] is a debugging tool for C programs with memory access errors. Purify uses "object code insertion" to instrument *all* memory accesses. The approach is similar to StackGuard, in that it does integrity checking of memory, but it does so on each memory access, rather than on each function return. As a result, Purify is both more general and more expensive than StackGuard, imposing a slowdown of 2 to 5 times the execution time of optimized code, making Purify more suitable for debugging software. StackGuard, in contrast, is intended to be left on for production use of the compiled code.

6.5 Type-Safe Languages

All of the vulnerabilities described here result from the lack of type safety in C. If the only operations that can be performed on a variable are those described by the type, then it is not possible to use creative input applied to variable `foo` to make arbitrary changes to the variable `bar`.

Type-safety is one of the foundations of the Java security model. Unfortunately, *errors* in the Java type checking system are one of the ways that Java programs and Java virtual machines can be attacked [5, 20]. If the correctness of the type checking system is in question, then programs depending on that type checking system for security benefit from these techniques in similar ways to the benefit provided to type-unsafe programs. Applying StackGuard techniques to Java programs and Java virtual machines may yield beneficial results.

7 Conclusions

We have presented StackGuard, a systematic compiler tool that prevents a broad class of buffer overflow security attacks from succeeding. We presented both security and performance analysis of the tool. Because the tool is oblivious to the specific attack and vulnerability being exploited, it is expected that this tool will also be able to stop buffer overflow attacks that have yet to be discovered, reducing the need for constant, rapid patching of software to stay secure.

In its most basic form, the tool requires only recompilation to make a program largely secure against buffer overflow attacks. In more elaborate forms, it provides an adaptive response to buffer overflow attacks, allowing systems to be configured to trade performance for survivability. We concluded with discussion on how to generalize these techniques to other areas of security vulnerability.

8 Availability

StackGuard is a small set of patches to `gcc`. We are releasing StackGuard under the Gnu Public License, while retaining copyright to OGI. StackGuard is available both as a patch to `gcc 2.7.2.2`, and as a complete `tar` file, at this location: `http://www/cse.ogi.edu/DISC/projects/immunix/StackGuard/`.

References

[1] Crispin Cowan, Tito Autrey, Charles Krasic, Calton Pu, and Jonathan Walpole. Fast Concurrent Dynamic Linking for an Adaptive Operating System. In *International Conference on Configurable Distributed Systems (ICCDS'96)*, Annapolis, MD, May 1996.

[2] Crispin Cowan, Andrew Black, Charles Krasic, Calton Pu, Jonathan Walpole, Charles Consel, and Eugen-Nicolae Volanschi. Specialization Classes: An Object Framework for Specialization. In *Proceedings of the Fifth International Workshop on Object-Orientation in Operating Systems (IWOOOS '96)*, Seattle, WA, October 27-28 1996.

[3] Crispin Cowan, Dylan McNamee, Andrew Black, Calton Pu, Jonathan Walpole, Charles Krasic, Renaud Marlet, and Qian Zhang. A Toolkit for Specializing Production Operating System Code. Technical Report CSE-97-004, Dept. of Computer Science and Engineering, Oregon Graduate Institute, March 1997.

[4] Michele Crabb. Curmudgeon's Executive Summary. In Michele Crabb, editor, *The SANS Network Security Digest*. SANS, 1997. Contributing Editors: Matt Bishop, Gene Spafford, Steve Bellovin, Gene Schultz, Rob Kolstad, Marcus Ranum, Dorothy Denning, Dan Geer, Peter Neumann, Peter Galvin, David Harley, Jean Chouanard.

[5] Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 1996.

http://www.cs.princeton.edu/sip/pub/secure96.html.

[6] "Solar Designer". Non-Executable User Stack. http://www.false.com/security/linux-stack/.

[7] D. Farmer. The COPS Security Checker System. In *Summer 1990 USENIX Conference*, page 165, Anaheim, CA, June 1990. http://www.trouble.org/cops/.

[8] Stephanie Forrest, Anil Somayaji, and David. H. Ackley. Building Diverse Computer Systems . In *HotOS-VI*, May 1997.

[9] Virgil Gligor, Serban Gavrila, and Sabari Gupta. Penetration Analysis Tools. Personal Communications, July 1997.

[10] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter USENIX Conference*, 1992. http://www.rational.com/support/techpapers/fast_detection/.

[11] Immunix. Adaptive System Survivability. http://www.cse.ogi.edu/DISC/projects/immunix, 1997.

[12] Richard Jones and Paul Kelly. Bounds Checking for C. http://www-ala.doc.ic.ac.uk/~phjk/BoundsChecking.html, July 1995.

[13] Barton P. Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz Revisited: A re-examination of the Reliability of UNIX Utilities and Services. Report, University of Wisconsin, 1995.

[14] B.P. Miller, L. Fredrikson, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):33–44, December 1990.

[15] "Mudge". How to Write Buffer Overflows. http://l0pht.com/advisories/bufero.html, 1997.

[16] "Aleph One". Bugtraq Mailing List. http://geek-girl.com/bugtraq/.

[17] "Aleph One". Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996.

[18] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[19] Calton Pu, Andrew Black, Crispin Cowan, Jonathan Walpole, and Charles Consel. Microlanguages for Operating System Specialization. In *SIGPLAN Workshop on Domain-Specific Languages*, Paris, France, January 1997.

[20] Jim Roskind. Panel: Security of Downloadable Executable Content. NDSS (Network and Distributed System Security), February 1997.

[21] Nathan P. Smith. Stack Smashing vulnerabilities in the UNIX Operating System. http://millcomm.com/~nate/machines/security/stack-smashing/nate-buffer.ps, 1997.

[22] Alexander Snarskii. FreeBSD Stack Integrity Patch. ftp://ftp.lucky.net/pub/unix/local/libc-letter, 1997.

[23] E. Spafford. The Internet Worm Program: Analysis. *Computer Communication Review*, January 1989.

[24] Eugen N. Volanschi, Charles Consel, Gilles Muller, and Crispin Cowan. Declarative Specialization of Object-Oriented Programs. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'97)*, Atlanta, GA, October 1997.

Data Mining Approaches for Intrusion Detection*

Wenke Lee Salvatore J. Stolfo

Computer Science Department
Columbia University
500 West 120th Street, New York, NY 10027
{wenke,sal}@cs.columbia.edu

Abstract

In this paper we discuss our research in developing general and systematic methods for intrusion detection. The key ideas are to use data mining techniques to discover consistent and useful patterns of system features that describe program and user behavior, and use the set of relevant system features to compute (inductively learned) classifiers that can recognize anomalies and known intrusions. Using experiments on the *sendmail* system call data and the network *tcpdump* data, we demonstrate that we can construct concise and accurate classifiers to detect anomalies. We provide an overview on two general data mining algorithms that we have implemented: the association rules algorithm and the frequent episodes algorithm. These algorithms can be used to compute the intra- and inter- audit record patterns, which are essential in describing program or user behavior. The discovered patterns can guide the audit data gathering process and facilitate feature selection. To meet the challenges of both efficient learning (mining) and real-time detection, we propose an agent-based architecture for intrusion detection systems where the learning agents continuously compute and provide the updated (detection) models to the detection agents.

1 Introduction

As network-based computer systems play increasingly vital roles in modern society, they have become the targets of our enemies and criminals. Therefore, we need to find the best ways possible to protect our systems.

The security of a computer system is compromised when an intrusion takes place. An intrusion can be defined [HLMS90] as "any set of actions that attempt to compromise the integrity, confidentiality or availability of a resource". Intrusion prevention techniques, such as user authentication (e.g. using passwords or biometrics), avoiding programming errors, and information protection (e.g., encryption) have been used to protect computer systems as a first line of defense. Intrusion prevention alone is not sufficient because as systems become ever more complex, there are always exploitable weakness in the systems due to design and programming errors, or various "socially engineered" penetration techniques. For example, after it was first reported many years ago, exploitable "buffer overflow" still exists in some recent system software due to programming errors. The policies that balance convenience versus strict control of a system and information access also make it impossible for an operational system to be completely secure.

Intrusion detection is therefore needed as another wall to protect computer systems. The elements central to intrusion detection are: *resources* to be protected in a target system, i.e., user accounts, file systems, system kernels, etc; *models* that characterize the "normal" or "legitimate" behavior of these resources; *techniques* that compare the actual system activities with the established models, and identify those that are "abnormal" or "intrusive".

Many researchers have proposed and implemented different models which define different measures of system behavior, with an ad hoc presumption that normalcy and anomaly (or illegitimacy) will be accurately manifested in the chosen set of system features that are modeled and measured. Intrusion detection techniques can be categorized into *misuse detection*, which uses patterns of well-known attacks or weak spots of the system to identify intrusions; and *anomaly detection*, which tries to determine whether deviation from the established normal us-

*This research is supported in part by grants from DARPA (F30602-96-1-0311) and NSF (IRI-96-32225 and CDA-96-25374)

age patterns can be flagged as intrusions.

Misuse detection systems, for example [KS95] and STAT [IKP95], encode and match the sequence of "signature actions" (e.g., change the ownership of a file) of known intrusion scenarios. The main shortcomings of such systems are: known intrusion patterns have to be hand-coded into the system; they are unable to detect any future (unknown) intrusions that have no matched patterns stored in the system.

Anomaly detection (sub)systems, such as IDES [LTG$^+$92], establish normal usage patterns (profiles) using statistical measures on system features, for example, the CPU and I/O activities by a particular user or program. The main difficulties of these systems are: intuition and experience is relied upon in selecting the system features, which can vary greatly among different computing environments; some intrusions can only be detected by studying the sequential interrelation between events because each event alone may fit the profiles.

Our research aims to eliminate, as much as possible, the manual and ad-hoc elements from the process of building an intrusion detection system. We take a data-centric point of view and consider intrusion detection as a data analysis process. Anomaly detection is about finding the normal usage patterns from the audit data, whereas misuse detection is about encoding and matching the intrusion patterns using the audit data. The central theme of our approach is to apply data mining techniques to intrusion detection. Data mining generally refers to the process of (automatically) extracting models from large stores of data [FPSS96]. The recent rapid development in data mining has made available a wide variety of algorithms, drawn from the fields of statistics, pattern recognition, machine learning, and database. Several types of algorithms are particularly relevant to our research:

Classification: maps a data item into one of several predefined categories. These algorithms normally output "classifiers", for example, in the form of decision trees or rules. An ideal application in intrusion detection will be to gather sufficient "normal" and "abnormal" audit data for a user or a program, then apply a classification algorithm to learn a classifier that will determine (future) audit data as belonging to the normal class or the abnormal class;

Link analysis: determines relations between fields in the database. Finding out the correlations in audit data will provide insight for selecting the right set of system features for intrusion detection;

Sequence analysis: models sequential patterns. These algorithms can help us understand what (time-based) sequence of audit events are frequently encountered together. These frequent event patterns are important elements of the behavior profile of a user or program.

We are developing a systematic framework for designing, developing and evaluating intrusion detection systems. Specifically, the framework consists of a set of environment-independent guidelines and programs that can assist a system administrator or security officer to

- select appropriate system features from audit data to build models for intrusion detection;
- architect a hierarchical detector system from component detectors;
- update and deploy new detection systems as needed.

The key advantage of our approach is that it can automatically generate concise and accurate detection models from large amount of audit data. The methodology itself is general and mechanical, and therefore can be used to build intrusion detection systems for a wide variety of computing environments.

The rest of the paper is organized as follows: Section 2 describes our experiments in building classification models for *sendmail* and network traffic. Section 3 presents the association rules and frequent episodes algorithms that can be used to compute a set of patterns from audit data. Section 4 briefly highlights the architecture of our proposed intrusion detection system. Section 5 outlines our future research plans.

2 Building Classification Models

In this section we describe in detail our experiments in constructing classification models for anomaly detection. The first set of experiments, first reported in [LSC97], is on the *sendmail* system call data, and the second is on the network *tcpdump* data.

2.1 Experiments on *sendmail* Data

There have been a lot of attacks on computer systems that are carried out as exploitations of the design and programming errors in privileged programs, those that can run as root. For example, a flaw in the *finger* daemon allows the attacker to use "buffer overflow" to trick the program to execute his malicious code. Recent research efforts by Ko et al. [KFL94] and Forrest et al. [FHSL96] attempted to build intrusion detection systems that monitor the execution of privileged programs and detect the attacks on their vulnerabilities. Forrest et al. discovered that the short sequences of system calls made by a program during its normal executions are very consistent, yet different from the sequences of its abnormal (exploited) executions as well as the executions of other programs. Therefore a database containing these normal sequences can be used as the "self" definition of the normal behavior of a program, and as the basis to detect anomalies. Their findings motivated us to search for simple and accurate intrusion detection models.

Stephanie Forrest provided us with a set of traces of the *sendmail* program used in her experiments [FHSL96]. We applied machine learning techniques to produce classifiers that can distinguish the exploits from the normal runs.

2.1.1 The *sendmail* System Call Traces

The procedure of generating the *sendmail* traces were detailed in [FHSL96]. Briefly, each file of the trace data has two columns of integers, the first is the process ids and the second is the system call "numbers". These numbers are indices into a lookup table of system call names. For example, the number "5" represents system call *open*. The set of traces include:

Normal traces: a trace of the *sendmail* daemon and a concatenation of several invocations of the *sendmail* program;

Abnormal traces: 3 traces of the *sscp* (*sunsendmailcp*) attacks, 2 traces of the *syslog-remote* attacks, 2 traces of the *syslog-local* attacks, 2 traces of the *decode* attacks, 1 trace of the *sm5x* attack and 1 trace of the *sm565a* attack. These are the traces of (various kinds of) abnormal runs of the *sendmail* program.

2.1.2 Learning to Classify System Call Sequences

In order for a machine learning program to learn the classification models of the "normal" and "abnormal" system call sequences, we need to supply it with a set of training data containing pre-labeled "normal" and "abnormal" sequences. We use a sliding window to scan the normal traces and create a list of unique sequences of system calls. We call this list the "normal" list. Next, we scan each of the intrusion traces. For each sequence of system calls, we first look it up in the normal list. If an exact match can be found then the sequence is labeled as "normal". Otherwise it is labeled as "abnormal" (note that the data gathering process described in [FHSL96] ensured that the normal traces include nearly all possible "normal" short sequences of system calls, as new runs of *sendmail* failed to generate new sequences). Needless to say all sequences in the normal traces are labeled as "normal". See Table 1 for an example of the labeled sequences. It should be noted that an intrusion trace contains many normal sequences in addition to the abnormal sequences since the illegal activities only occur in some places within a trace.

System Call Sequences (length 7)	Class Labels
4 2 66 66 4 138 66	"normal"
...	...
5 5 5 4 59 105 104	"abnormal"
...	...

Table 1: Pre-labeled System Call Sequences of Length 7

We applied RIPPER [Coh95], a rule learning program, to our training data. The following learning tasks were formulated to induce the rule sets for normal and abnormal system call sequences:

- Each record has n positional attributes, p_1, p_2, ..., p_n, one for each of the system calls in a sequence of length n; plus a class label, "normal" or "abnormal"
- The training data is composed of normal sequences taken from 80% of the normal traces, plus the abnormal sequences from 2 traces of the *sscp* attacks, 1 trace of the *syslog-local* attack, and 1 trace of the *syslog-remote* attack
- The testing data includes both normal and abnormal traces not used in the training data.

RIPPER outputs a set of if-then rules for the "minority" classes, and a default "true" rule for the remaining class. The following exemplar RIPPER rules were generated from the system call data:

> normal:- $p_2 = 104, p_7 = 112$. [meaning: if p_2 is 104 ($vtimes$) and p_7 is 112 ($vtrace$) then the sequence is "normal"]

normal:- $p_6 = 19, p_7 = 105$. [meaning: if p_6 is 19 (*lseek*) and p_7 is 105 (*sigvec*) then the sequence is "normal"]

...

abnormal:- true. [meaning: if none of the above, the sequence is "abnormal"]

These RIPPER rules can be used to predict whether a sequence is "abnormal" or "normal". But what the intrusion detection system needs to know is whether the trace being analyzed is an intrusion or not. We use the following post-processing scheme to detect whether a given trace is an intrusion based on the RIPPER predictions of its constituent sequences:

1. Use a sliding window of length $2l+1$, e.g., 7, 9, 11, 13, etc., and a sliding (shift) step of l, to scan the predictions made by the RIPPER rules on system call sequences.
2. For each of the (length $2l + 1$) regions of RIPPER predictions generated in Step 1, if more than l predictions are "abnormal" then the current region of predictions is an "abnormal" region. (Note that l is an input parameter).
3. If the percentage of abnormal regions is above a threshold value, say 2%, then the trace is an intrusion.

This scheme is an attempt to filter out the spurious prediction errors. The intuition behind this scheme is that when an intrusion actually occurs, the majority of adjacent system call sequences are abnormal; whereas the prediction errors tend to be isolated and sparse. In [FHSL96], the percentage of the mismatched sequences (out of the total number of matches (lookups) performed for the trace) is used to distinguish normal from abnormal. The "mismatched" sequences are the abnormal sequences in our context. Our scheme is different in that we look for abnormal regions that contain more abnormal sequences than the normal ones, and calculate the percentage of abnormal regions (out of the total number of regions). Our scheme is more sensitive to the temporal information, and is less sensitive to noise (errors).

RIPPER only outputs rules for the "minority" class. For example, in our experiments, if the training data has fewer abnormal sequences than the normal ones, the output RIPPER rules can be used to identify abnormal sequences, and the default (everything else) prediction is normal. We conjectured that a set of specific rules for normal sequences can be used as the "identity" of a program, and thus can be used to detect any known and unknown intrusions (anomaly intrusion detection). Whereas having only the rules for abnormal sequences only gives us the capability to identify known intrusions (misuse intrusion detection).

	% abn.	% abn. in experiment			
Traces	[FHSL96]	A	B	C	D
sscp-1	5.2	41.9	32.2	40.0	33.1
sscp-2	5.2	40.4	30.4	37.6	33.3
sscp-3	5.2	40.4	30.4	37.6	33.3
syslog-r-1	5.1	30.8	21.2	30.3	21.9
syslog-r-2	1.7	27.1	15.6	26.8	16.5
syslog-l-1	4.0	16.7	11.1	17.0	13.0
syslog-l-2	5.3	19.9	15.9	19.8	15.9
decode-1	0.3	4.7	2.1	3.1	2.1
decode-2	0.3	4.4	2.0	2.5	2.2
sm565a	0.6	11.7	8.0	1.1	1.0
sm5x	2.7	17.7	6.5	5.0	3.0
sendmail	0	1.0	0.1	0.2	0.3

Table 2: Comparing Detection of Anomalies. The column [FHSL96] is the percentage of the abnormal sequences of the traces. Columns A, B, C, and D are the percentages of abnormal regions (as measured by the post-processing scheme) of the traces. *sendmail* is the 20% normal traces not used in the training data. Traces in bold were included in the training data, the other traces were used as testing data only.

We compare the results of the following experiments that have different distributions of abnormal versus normal sequences in the training data:

Experiment A: 46% normal and 54% abnormal, sequence length is 11;

Experiment B: 46% normal and 54% abnormal, sequence length is 7;

Experiment C: 46% abnormal and 54% normal, sequence length is 11;

Experiment D: 46% abnormal and 54% normal, sequence length is 7.

Table 2 shows the results of using the classifiers from these experiments to analyze the traces. We report here the percentage of abnormal regions (as measured by our post-processing scheme) of each trace, and compare our results with Forrest et al., as reported in [FHSL96]. From Table 2, we can see that in general, intrusion traces generate much larger percentages of abnormal regions than the normal traces. We call these measured percentages the "scores" of the traces. In order to establish a threshold score for identifying intrusion traces, it is desirable that there is a sufficiently large gap between the scores of the normal sendmail traces and the low-end scores of the intrusion traces. Comparing experiments that used the same sequence length, we observe that such a gap in A, 3.4, is larger than the gap in C, 0.9; and 1.9 in B is larger than 0.7 in D. The RIPPER rules from experiments A and B describe the patterns of the normal sequences. Here the results show that these rules can be used to identify the intrusion traces, including those not seen in the training data, namely, the *decode* traces, the *sm565a* and *sm5x* traces. This confirms our conjecture that rules for normal patterns can be used for anomaly detection. The RIPPER rules from experiments C and D specify the patterns of abnormal sequences in the intrusion traces included in the training data. The results indicate that these rules are very capable of detecting the intrusion traces of the "known" types (those seen in the training data), namely, the *sscp-3* trace, the *syslog-remote-2* trace and the *syslog-local-2* trace. But comparing with the rules from A and B, the rules in C and D perform poorly on intrusion traces of "unknown" types. This confirms our conjecture that rules for abnormal patterns are good for misuse intrusion detection, but may not be as effective in detecting future ("unknown") intrusions.

The results from Forrest et al. showed that their method required a very low threshold in order to correctly detect the *decode* and $sm565a$ intrusions. While the results here show that our approach generated much stronger "signals" of anomalies from the intrusion traces, it should be noted that their method used all of the normal traces but not any of the intrusion traces in training.

2.1.3 Learning to Predict System Calls

Unlike the experiments in Section 2.1.2 which required abnormal traces in the training data, here we wanted to study how to compute an anomaly detector given just the normal traces. We conducted experiments to learn the (normal) correlation among system calls: the nth system calls or the middle system calls in (normal) sequences of length n.

The learning tasks were formulated as follows:

- Each record has $n-1$ positional attributes, p_1, p_2, $\ldots$, p_{n-1}, each being a system call; plus a class label, the system call of the nth position or the middle position
- The training data is composed of (normal) sequences taken from 80% of the normal sendmail traces
- The testing data is the traces not included in the training data, namely, the remaining 20% of the normal sendmail traces and all the intrusion traces.

RIPPER outputs rules in the following form:

> 38 :- $p_3 = 40$, $p_4 = 4$. [meaning: if p_3 is 40 ($lstat$) and p_4 is 4 ($write$), then the 7th system call is 38 ($stat$).]
>
> ...
>
> 5:- true. [meaning: if none of the above, then the 7th system calls is 5 ($open$).]

Each of these RIPPER rules has some "confidence" information: the number of matched examples (records that conform to the rule) and the number of unmatched examples (records that are in conflict with the rule) in the training data. For example, the rule for "38 ($stat$)" covers 12 matched examples and 0 unmatched examples. We measure the confidence value of a rule as the number of matched examples divided by the sum of matched and unmatched examples. These rules can be used to analyze a trace by examining each sequence of the trace. If a violation occurs (the actual system call is not the same as predicted by the rule), the "score" of the trace is incremented by 100 times the confidence of the violated rule. For example, if a sequence in the trace has $p_3 = 40$ and $p_4 = 4$, but $p_7 = 44$ instead of 38, the total score of the trace is incremented by 100 since the confidence value of this violated rule is 1. The averaged score (by the total number of sequences) of the trace is then used to decide whether an intrusion has occurred.

Table 3 shows the results of the following experiments:

Experiment A: predict the 11th system call;

Experiment B: predict the middle system call in a sequence of length 7;

Experiment C: predict the middle system call in a sequence of length 11;

Experiment D: predict the 7th system call.

We can see from Table 3 that the RIPPER rules from experiments A and B are effective because the gap between the score of normal sendmail and the low-end scores of intrusion traces, 3.9, and 3.3 respectively, are large enough. However, the rules from C and D perform poorly. Since C predicts the middle system call of a sequence of length 11 and D predicts the 7th system call, we reason that the training data (the normal traces) has no stable patterns for the 6th or 7th position in system call sequences.

	averaged score of violations			
Traces	Exp. A	Exp. B	Exp. C	Exp. D
sscp-1	24.1	13.5	14.3	24.7
sscp-2	23.5	13.6	13.9	24.4
sscp-3	23.5	13.6	13.9	24.4
syslog-r-1	19.3	11.5	13.9	24.0
syslog-r-2	15.9	8.4	10.9	23.0
syslog-l-1	13.4	6.1	7.2	19.0
syslog-l-2	15.2	8.0	9.0	20.2
decode-1	9.4	3.9	2.4	11.3
decode-2	9.6	4.2	2.8	11.5
sm565a	14.4	8.1	9.4	20.6
sm5x	17.2	8.2	10.1	18.0
sendmail	5.7	0.6	1.2	12.6

Table 3: Detecting Anomalies using Predicted System Calls. Columns A, B, C, and D are the averaged scores of violations of the traces. *sendmail* is the 20% normal traces not used in the training data. None of the intrusion traces was used in training.

2.1.4 Discussion

Our experiments showed that the normal behavior of a program execution can be established and used to detect its anomalous usage. This confirms the results of other related work in anomaly detection. The weakness of the model in [FHSL96] may be that the recorded (rote learned) normal sequence database may be too specific as it contains $\sim 1,500$ entries. Here we show that a machine learning program, RIPPER, was able to generalize the system call sequence information, from 80% of the normal sequences, to a set of concise and accurate rules (the rule sets have 200 to 280 rules, and each rule has 2 or 3 attribute tests). We demonstrated that these rules were able to identify unseen intrusion traces as well as normal traces.

We need to search for a more predictive classification model so that the anomaly detector has higher confidence in flagging intrusions. Improvement in accuracy can come from adding more features, rather than just the system calls, into the models of program execution. For example, the directories and the names of the files touched by a program can be used. In [Fra94], it is reported that as the number of features increases from 1 to 3, the classification error rate of their network intrusion detection system decreases dramatically. Furthermore, the error rate stabilizes after the size of the feature set reaches 4, the optimal size in their experiments. Many operating systems provide auditing utilities, such as the BSM audit of Solaris, that can be configured to collect abundant information (with many features) of the activities in a host system. From the audit trails, information about a process (program) or a user can then be extracted. The challenge now is to efficiently compute accurate patterns of programs and users from the audit data.

A key assumption in using a learning algorithm for anomaly detection (and to some degree, misuse detection) is that the training data is nearly "complete" with regard to all possible "normal" behavior of a program or user. Otherwise, the learned detection model can not confidently classify or label an unmatched data as "abnormal" since it can just be an unseen "normal" data. For example, the experiments in Section 2.1.3 used 80% of "normal" system call sequences; whereas the experiments in Section 2.1.2 actually required all "normal" sequences in order to pre-label the "abnormal" sequences to create the training data. During the audit data gathering process, we want to ensure that as much different normal behavior as possible is captured. We first need to have a simple and incremental (continuously learning) summary measure of an audit trail so that we can update this measure as each new audit trail is processed, and can stop the audit process when the measure stabilizes. In Section 3, we propose to use the frequent intra- and inter- audit record patterns as the summary measure of an audit trail, and describe the algorithms to compute these patterns.

2.2 Experiments on *tcpdump* Data

There are two approaches for network intrusion detection: one is to analyze the audit data on each host of the network and correlate the evidence from the hosts. The other is to monitor the network traffic directly using a packet capturing program such as *tcpdump* [JLM89]. In this section, we describe how classifiers can be induced from *tcpdump* data to distinguish network attacks from normal traffic.

2.2.1 The *tcpdump* Data

We obtained a set of *tcpdump* data, available via http at "iris.cs.uml.edu:8080/network.html", that is part of an Information Exploration Shootout (see "http://iris.cs.uml.edu:8080"). *tcpdump* was executed on the gateway that connects the enterprise LAN and the external networks. It captured the headers (not the user data) of the network packets that passed by the network interface of the gateway. Network traffic between the enterprise LAN and external networks, as well as the broadcast packets within the LAN were therefore collected. For the purposes of the shootout, filters were used so that *tcpdump* only collected Internet Transmission Control Protocol (TCP) and Internet User Datagram Protocol (UDP) packets. The data set consists of 3 runs of *tcpdump* on generated network intrusions[1] and one *tcpdump* run on normal network traffic (with no intrusions). The output of each *tcpdump* run is in a separate file. The traffic volume (number of network connections) of these runs are about the same. Our experiments focused on building an anomaly detection model from the normal dataset.

Since *tcpdump* output is not intended specifically for security purposes, we had to go through multiple iterations of data pre-processing to extract meaningful features and measures. We studied TCP/IP and its security related problems, for example [Ste84, Pax97, ABH$^+$96, Pax98, Bel89, PV98], for guidelines on the protocols and the important features that characterize a connection.

2.2.2 Data Pre-processing

We developed a script to scan each *tcpdump* data file and extract the "connection" level information about the network traffic. For each TCP connection, the script processes packets between the two ports of the participating hosts, and:

- checks whether 3-way handshake has been properly followed to establish the connection. The following errors are recorded: connection rejected, connection attempted but not established (the initiating host never receives a SYN acknowledgment), and unwanted SYN acknowledgment received (no connection request, a SYN packet, was sent first),
- monitors each data packet and ACK packet, keeps a number of counters in order to calculate these statistics of the connection: resent rate, wrong resent rate, duplicate ACK rate, hole rate, wrong (data packet) size rate, (data) bytes sent in each direction, percentage of data packet, and percentage of control packet, and
- watches how connection is terminated: normal (both sides properly send and receive FINs), abort (one host sends RST to terminate, and all data packets are properly ACKed), half closed (only one host sends FIN), and disconnected.

Since UDP is connectionless (no connection state), we simply treat each packet as a connection.

A connection record, in preparation of data mining, now has the following fields (features): start time, duration, participating hosts, ports, the statistics of the connection (e.g., bytes sent in each direction, resent rate, etc.), flag ("normal" or one of the recorded connection/termination errors), and protocol (TCP or UDP). From the ports, we know whether the connection is to a well-known service, e.g., *http* (port 80), or a user application.

We call the host that initiates the connection, i.e., the one that sends the first SYN, as the source, and the other as the destination. Depending on the direction from the source to the destination, a connection is in one of the three types: *out-going* - from the LAN to the external networks; *in-coming* - from the external networks to the LAN; and *inter-LAN* - within the LAN. Taking the topologies of the network into consideration is important in network intrusion detection. Intuitively, intrusions (which come from outside) may first exhibit some abnormal patterns (e.g., penetration attempts) in the *in-coming* connections, and subsequently in the *inter-LAN* (e.g., doing damage to the LAN) and/or the *out-going* (e.g., stealing/uploading data) connections. Analyzing these types of connections and constructing corresponding detection models separately may improve detection accuracy.

2.2.3 Experiments and Results

For each type (direction) of the connections, we formulated the classification experiments as the following:

- Each (connection) record uses the destination service (port) as the class label, and all the other connection features as attributes;
- The training data is 80% of the connections from the normal *tcpdump* data file, while the test data includes the remaining 20% from the normal *tcpdump* data file, and all the connections from the 3 *tcpdump* data files marked as having embedded attacks;
- 5-fold cross validation evaluation is reported here. The process (training and testing) is repeated 5 times, each time using a different 80% of the normal data as the training data (and accordingly the

[1] Note that, to this date, the organizers of the shootout have not provided us with information, i.e., the times, targets, and actions, of these network intrusions.

	% misclassification (by traffic type)		
Data	out-going	in-coming	inter-LAN
normal	3.91%	4.68%	4%
intrusion1	3.81%	6.76%	22.65%
intrusion2	4.76%	7.47%	8.7%
intrusion3	3.71%	13.7%	7.86%

Table 4: Misclassification Rate on Normal and Intrusion Data. Separate classifiers were trained and tested on connection data of each traffic type. "normal" is the 20% data set aside from the training data. No intrusion data was used for training.

different remaining 20% of the normal data as part of the test data), and the averaged accuracy of the classifiers from the 5 runs is reported.

We again applied RIPPER to the connection data. The resulting classifier characterizes the normal patterns of each service in terms of the connection features. When using the classifier on the testing data, the percentage of misclassifications on each *tcpdump* data set is reported. Here a misclassification is the situation where the the classifier predicts a destination service (according to the connection features) that is different from the actual. This misclassification rate should be very low for normal connection data and high for intrusion data. The intuition behind this classification model is straightforward: when intrusions take place, the features (characteristics) of connections to certain services, for example, *ftp*, are different from the normal traffic patterns (of the same service).

The results from the first round of experiments, as shown in Table 4, were not very good: the differences in the misclassification rates of the normal and intrusion data were small, except for the *inter-LAN* traffic of some intrusions.

We then redesigned our set of features by adding some continuous and intensity measures into each connection record:

- Examining all connections in the past n seconds, and counting the number of: connection establishment errors (e.g., "connection rejected"), all other types of errors (e.g., "disconnected"), connections to designated system services (e.g., *ftp*), connections to user applications, and connections to the same service as the current connection;
- Calculate for the past n seconds, the per-connection average duration and data bytes (on both directions) of all connections, and the same averages of connections to the same service.

	% misclassification (by traffic type)		
Data	out-going	in-coming	inter-LAN
normal	0.88%	0.31%	1.43%
intrusion1	2.54%	27.37%	20.48%
intrusion2	3.04%	27.42%	5.63%
intrusion3	2.32%	42.20%	6.80%

Table 5: Using Temporal-Statistical Measures to Improve Classification Accuracy. Here the time interval is 30 seconds.

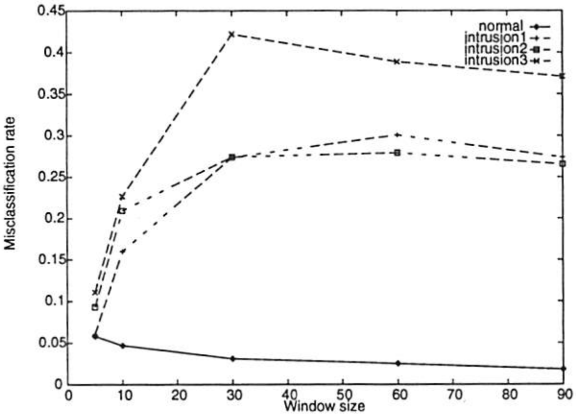


Figure 1: Effects of Window Sizes on Misclassification Rates

These additional temporal-statistical features provide additional information of the network activity from a continuous perspective, and provide more insight into anomalies. For example, a low rate of error due to innocent attempts and network glitches in a short time span is expected, but an excess beyond the (averaged) norm indicates anomalous activity. Table 5 shows the improvement of adding these features. Here, using a time interval of 30 seconds (i.e., $n = 30s$), we see that the misclassification rates on the intrusion data are much higher than the normal data, especially for the *in-coming* traffic. The RIPPER rule set (the classifier) has just 9 rules and 25 conditions. For example, one rule says "if the average number of bytes from source to destination (of the connections to the same service) is 0, and the percentage of control packets in the current connection is 100%, then the service is *auth*".

To understand the effects of the time intervals on the misclassification rates, we ran the experiments using various time intervals: 5s, 10s, 30s, 60s, and 90s. The effects on the *out-going* and *inter-LAN* traffic were very small. However, as Figure 1 shows, for the *in-coming* traffic, the misclassification rates on the intrusion data increase dramatically as the time interval goes from 5s to 30s, then stabilizes or tapers off afterwards.

2.2.4 Discussion

We learned some important lessons from the experiments on the *tcpdump* data. First, when the collected data is not designed specifically for security purposes or can not be used directly to build a detection model, a considerable amount of (iterative) data pre-processing is required. This process fundamentally requires a lot of domain knowledge, and may not be easily automated. Second, in general, adding temporal-statistical features can improve the accuracy of the classification model.

There are also much needed improvements to our current approach: First, deciding upon the right set of features is difficult and time consuming. For example, many trials were attempted before we came up with the current set of features and time intervals. We need useful tools that can provide insight into the patterns that may be exhibited in the data. Second, we should provide tools that can help administrative staff understand the nature of the anomalies.

2.3 Combining Multiple Classifiers

The classifiers described in this section each models a single aspect of the system behavior. They are what we call the base (single level) classifiers. Combining evidence from multiple base classifiers that each models different aspect of the target system is likely to improve the effectiveness in detecting intrusions. For example, in addition to the classifier for network traffic (using *tcpdump* data), we can include the classifiers on the commands issued during the (connection) sessions of well-known services, e.g. *ftp*, *telnet* etc. The combined evidence of anomalous traffic patterns and session behavior leads to a more accurate assertion that the network is under attack. A priority in our research plan is to study and experiment with (inductively learned) classification models that combine evidence from multiple (base) detection models. The general approach in learning such a meta-detection model can be summarized as follows:

- Build base classifiers that each models different aspect of the target system;
- Formulate the meta learning task: each record in the training data is a collection of the evidence (generated at the same time period) from the base classifiers; each attribute value in a record is 1 or 0, the prediction (evidence) from a base classifier that the modeled behavior is "normal" or "abnormal" (i.e., it fits the model or not).
- Apply a learning algorithm to produce the meta classifier.

The meta detection model is actually a hierarchy of detection models. At the bottom, the base classifiers take audit data as input and output evidence to the meta classifier, which in turn outputs the final assertion.

Our research activities in JAM [SPT$^+$97], which focus on the accuracy and efficiency of meta classifiers, will contribute significantly to our effort in building meta detection models.

3 Mining Patterns from Audit Data

In order to construct an accurate (effective) base classifier, we need to gather a sufficient amount of training data and identify a set of meaningful features. Both of these tasks require insight into the nature of the audit data, and can be very difficult without proper tools and guidelines. In this section we describe some algorithms that can address these needs. Here we use the term "audit data" to refer to general data streams that have been properly processed for detection purposes. An example of such data streams is the connection record data extracted from the raw *tcpdump* output.

3.1 Association Rules

The goal of mining association rules is to derive multi-feature (attribute) correlations from a database table. A simple yet interesting commercial application of the association rules algorithm is to determine what items are often purchased together by customers, and use that information to arrange store layout. Formally, given a set of records, where each record is a set of items, an association rule is an expression $X \Rightarrow Y, confidence, support$ [SA95]. X and Y are subsets of the items in a record, *support* is the percentage of records that contain $X + Y$, whereas *confidence* is $\frac{support(X+Y)}{support(X)}$. For example, an association rule from the shell command history file (which is a stream of commands and their arguments) of a user is

$$trn \Rightarrow rec.humor; [0.3, 0.1],$$

which indicates that 30% of the time when the user invokes trn, he or she is reading the news in $rec.humor$, and reading this newsgroup accounts for 10% of the activities recorded in his or her command history file. Here 0.3 is the *confidence* and 0.1 is the *support*.

The motivation for applying the association rules algorithm to audit data are:

- Audit data can be formatted into a database table where each row is an audit record and each column is a field (system feature) of the audit records;
- There is evidence that program executions and user activities exhibit frequent correlations among system features. For example, one of the reasons that "program policies", which codify the access rights of privileged programs, are concise and capable to detect known attacks [KFL94] is that the intended behavior of a program, e.g., $read$ and $write$ files from certain directories with specific permissions, is very consistent. These consistent behaviors can be captured in association rules;
- We can continuously merge the rules from a new run to the aggregate rule set (of all previous runs).

Our implementation follows the general association rules algorithm, as described in [Sri96].

3.2 Frequent Episodes

While the association rules algorithm seeks to find intra-audit record patterns, the frequent episodes algorithm, as described in [MTV95], can be used to discover inter- audit record patterns. A frequent episode is a set of events that occur frequently within a time window (of a specified length). The events must occur (together) in at least a specified minimum frequency, min_fr, sliding time window. Events in a $serial$ episode must occur in partial order in time; whereas for a $parallel$ episode there is no such constraint. For X and Y where $X+Y$ is a frequent episode, $X \Rightarrow Y$ with $confidence = \frac{frequency(X+Y)}{frequency(X)}$ and $support = frequency(X+Y)$ is called a frequent episode rule. An example frequent serial episode rule from the log file of a department's Web site is

$$home, research \Rightarrow theory; [0.2, 0.05], [30s]$$

which indicates that when the home page and the research guide are visited (in that order), in 20% of the cases the theory group's page is visited subsequently within the same 30s time window, and this sequence of visits occurs 5% of the total (the 30s) time windows in the log file (that is, approximately 5% of all the records).

We seek to apply the frequent episodes algorithm to analyze audit trails since there is evidence that the sequence information in program executions and user commands can be used to build profiles for anomaly detection [FHSL96, LB97]. Our implementation followed the description in [MTV95].

3.3 Using the Discovered Patterns

The association rules and frequent episodes can be used to guide the audit process. We run a program many times and under different settings. For each new run, we compute its rule set (that consists of both the association rules and the frequent episodes) from the audit trail, and update the (existing) aggregate rule sets using the following *merge* process:

- For each rule in the new rule set: find a match in the aggregate rule set. A match is defined as the exact matches on both the LHS and RHS of the rules, plus ε matches (using ranges), on the $support$ (or $frequency$) and $confidence$ values
- If a match is found, increment the $match\_count$ of the matched rule in the aggregate rule set. Otherwise, add the new rule and initialize its $match\_count$ to be 1.

When the rule set stabilizes (there are no new rules added), we can stop the data gathering process since we have produced a near complete set of audit data for the normal runs. We then *prune* the rule set by eliminating the rules with low $match\_count$, according to a user-defined threshold on the ratio of $match\_count$ over the total number of audit trails. The system builders can then use the correlation information in this final *profile* rule set to select a subset of the relevant features for the classification tasks. We plan to build a support environment to integrate the process of user selection of features, computing a classifier (according to the feature set), and presenting the performance of the classifier. Such a support system can speed up the iterative feature selection process, and help ensure the accuracy of a detection model.

We believe that the discovered patterns from (the extensively gathered) audit data can be used directly for anomaly detection. We compute a set of association

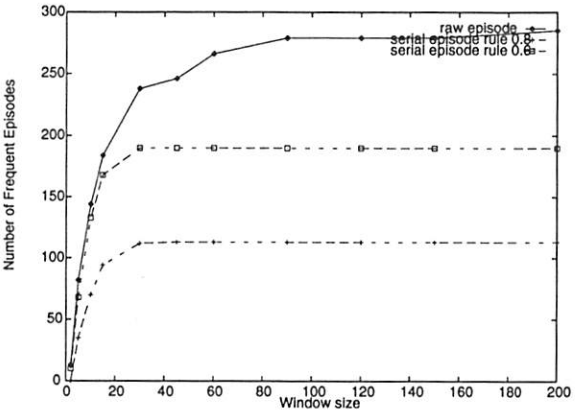


Figure 2: Effects of Window Sizes on the Number of Frequent Episodes.

rules and frequent episodes from a new audit trail, and compare it with the established $profile$ rule set. Scoring functions can be used to evaluate the deviation scores for: missing rules with high $support$, violation (same antecedent but different consequent) of rules with high $support$ and $confidence$, new (unseen) rules, and significant changes in $support$ of rules.

3.3.1 *tcpdump* Data Revisited

We ran some preliminary experiments using our association rules and frequent episodes programs on the *tcpdump* data that was used in the experiments described in Section 2.2.

We wanted to study how the frequent episodes algorithm can help us determine the time window used in gathering temporal-statistical features. We ran the algorithm on the "normal" *in-coming* connection records (without the temporal-statistical features). We set the program to produce two types of output: raw serial and parallel episodes (no rules were generated) and serial episode rules. For raw episodes, we used $min\_fr = 0.3$. And for serial episode rules, we used $min\_fr = 0.1$ and $min\_conf = 0.6$ and 0.8. We used different time window sizes (win): 2s, 5s, 10s, 15s, 30s, 45s, 60s, 90s, 120s, 150s, and 200s; and recorded the number of frequent episodes generated on each win. In Figure 2 we see that the number of frequent episodes (raw episodes or serial episode rules) increases sharply as win goes from 2s to 30s, it then gradually stabilizes (note that by the nature of the frequent episodes algorithm, the number of episodes can only increase as win increases). This phenomenon coincides with the trends in Figure 1. Note that here we made the particular choice of the parameters (i.e., min_fr, min_conf) only for the purpose of controlling the maximum size of the episode rule set. Different settings exhibited the same phenomenon. We conjecture (and will verify with further experiments on other data sets) that we can use this technique to analyze data streams and automatically discover the most important temporal measure: the time window size, i.e., the period of time within which to measure appropriate statistical features to maximize classifier accuracy. Intuitively, the first requirement of a time window size is that its set of sequence patterns is stable, that is, sufficient patterns are captured and noise is small.

We also ran both the association rules and frequent episodes programs on all the *in-coming* connection data, and compared the rule sets from the normal data with the intrusion data. The purpose of this experiment was to determine how these programs can provide insight into the (possible) patterns of intrusions. The frequent episodes generated were serial episode rules with $win = 30s$, $min\_fr = 0.1$ and $min\_conf = 0.8$. The associations rules were generated using $min\_support = 0.3$ and $min\_confidence = 0.9$. We manually examined and compared the rule sets to look for "unique" patterns that exist in the intrusion data (but not in the normal data). Here are some results:

intrusion1: the unique serial rules are related to "ftp-data as the source application", for example,

> *src_srv* = "ftp-data", *src_srv* = "user-apps" ⟹ *src_srv* = "ftp-data"; [0.96, 0.11], [30s]

This rule means: when a connection with a user application as the source service follows a connection with *ftp-data*, 96% of the cases, a connection with *ftp-data* follows and falls into the same time window (30s); and this patterns occur 11% of the time. The unique association rules are related to "destination service is a user application", for example,

> *dst_srv* = "user-apps" ⟹ *duration* = 0, *dst_to_src_bytes* = 0; [0.9, 0.33]

This rule means: when the destination service of a connection is a user application, 90% of the cases, the duration and the number of data bytes from the destination to the source are both 0; and this pattern occurs 33% of the time.

intrusion2: the results are nearly identical to $intrusion1$ in terms of the unique serial rules and association rules.

intrusion3: the unique serial rules are related to "auth as the destination service", for example,

> *dst_srv* = "auth" $\Longrightarrow$ *flag* = "unwanted_syn_ack"; [0.82, 0.1], [30s]

and

> *dst_srv* = "auth" $\Longrightarrow$ *dst_srv* = "user-apps", *dst_srv* = "auth"; [0.82, 0.1], [30s]

There are a significant number of unique association rules in regard to "smtp is the source application". Many of these rules suggest connection error of *smtp*, for example,

> *src_srv* = "smtp" $\Longrightarrow$ *duration* = 0, *flag* = "unwanted_syn_ack", *dst_srv* = "user-apps"; [1.0, 0.38]

These rules may provide hints about the intrusions. For example, the unique (not normal) serial episodes in $intrusion1$ and $intrusion2$ reveal that there are a large number of ftp data transfer activities; whereas the unique serial episodes in $intrusion3$ suggest that a large number of connections to the $auth$ service were attempted.

4 Architecture Support

The biggest challenge of using data mining approaches in intrusion detection is that it requires a large amount of audit data in order to compute the profile rule sets. And the fact that we may need to compute a detection model for each resource in a target system makes the data mining task daunting. Moreover, this learning (mining) process is an integral and continuous part of an intrusion detection system because the rule sets used by the detection module may not be static over a long period of time. For example, as a new version of a system software arrives, we need to update the "normal" profile rules. Given that data mining is an expensive process (in time and storage), and real-time detection needs to be lightweight to be practical, we can't afford to have a monolithic intrusion detection system.

We propose a system architecture, as shown in Figure 3, that includes two kinds of intelligent agents: the learning agents and the detection agents. A learning agent, which may reside in a server machine for its computing power, is responsible for computing and maintaining the rule sets for programs and users. It produces both the base detection models and the meta detection models. The task of a learning agent, to compute accurate models from very large amount of audit data, is an example of the "scale-up" problem in machine learning. We expect that our research in agent-based meta-learning systems [SPT$^+$97] will contribute significantly to the implementation of the learning agents. Briefly, we are studying how to partition and dispatch data to a host of machines to compute classifiers in parallel, and re-import the remotely learned classifiers and combine an accurate (final) meta-classifier, a hierarchy of classifiers [CS93].

A detection agent is generic and extensible. It is equipped with a (learned and periodically updated) rule set (i.e., a classifier) from the remote learning agent. Its detection engine "executes" the classifier on the input audit data, and outputs evidence of intrusions. The main difference between a base detection agent and the meta detection agent is: the former uses preprocessed audit data as input while the later uses the evidence from all the base detection agents. The base detection agents and the meta detection agent need not be running on the same host. For example, in a network environment, a meta agent can combine reports from (base) detection agents running on each host, and make the final assertion on the state of the network.

The main advantages of such a system architecture are:

- It is easy to construct an intrusion detection system as a compositional hierarchy of generic detection agents.
- The detection agents are lightweight since they can function independently from the heavyweight learning agents, in time and locale, so long as it is already equipped with the rule sets.
- A detection agent can report new instances of intrusions by transmitting the audit records to the learning agent, which can in turn compute an updated classifier to detect such intrusions, and dispatch them to all detection agents. Interestingly, the capability to derive and disseminate anti-virus codes faster than the virus can spread is also considered a key requirement for anti-virus systems [KSSW97].

5 Conclusion and Future Work

In this paper we proposed a systemic framework that employs data mining techniques for intrusion detection. This framework consists of classification, association rules, and frequence episodes programs, that can be used

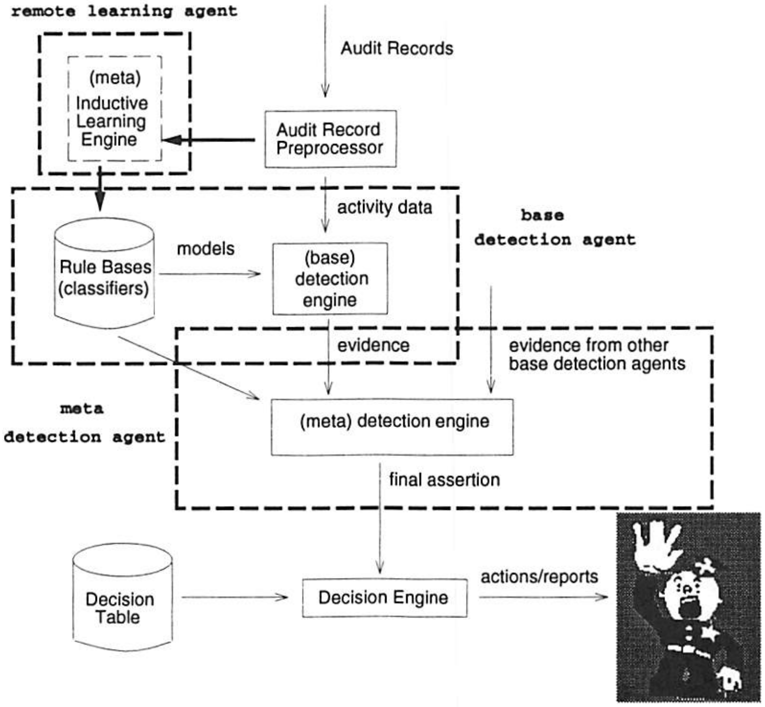


Figure 3: An Architecture for Agent-Based Intrusion Detection System

to (automatically) construct detection models. The experiments on *sendmail* system call data and network *tcpdump* data demonstrated the effectiveness of classification models in detecting anomalies. The accuracy of the detection models depends on sufficient training data and the right feature set. We suggested that the association rules and frequent episodes algorithms can be used to compute the consistent patterns from audit data. These frequent patterns form an abstract summary of an audit trail, and therefore can be used to: guide the audit data gathering process; provide help for feature selection; and discover patterns of intrusions. Preliminary experiments of using these algorithms on the *tcpdump* data showed promising results.

We are in the initial stages of our research, much remains to be done including the following tasks:

- Implement a support environment for system builders to iteratively drive the integrated process of pattern discovering, system feature selection, and construction and evaluation of detection models;
- Investigate the methods and benefits of combining multiple simple detection models. We need to use multiple audit data streams for experiments;
- Implement a prototype agent-based intrusion detection system. JAM [SPT+97] already provides a base infrastructure;
- Evaluate our approach using extensive audit data sets, some of which is presently under construction at Rome Labs.

6 Acknowledgments

We are very grateful to Stephanie Forrest and Steven A. Hofmeyr, both of University of New Mexico, for providing us with the system call data and explaining the details of their experiments. We also wish to thank Philip K. Chan of Florida Institute of Technology and David Wei Fan of Columbia University for their helpful advice.

References

[ABH+96] D. Atkins, P. Buis, C. Hare, R. Kelley, C. Nachenberg, A. B. Nelson, P. Phillips, T. Ritchey, and W. Steen. *Internet Security Professional Reference*. New Riders Publishing, 1996.

[Bel89] S. M. Bellovin. Security problems in the tcp/ip protocol suite. *Computer Communication Review*, 19(2):32–48, April 1989.

[Coh95] W. W. Cohen. Fast effective rule induction. In *Machine Learning: the 12th International Conference*, Lake Taho, CA, 1995. Morgan Kaufmann.

[CS93] P. K. Chan and S. J. Stolfo. Toward parallel and distributed learning by meta-learning. In *AAAI Workshop in Knowledge Discovery in Databases*, pages 227–240, 1993.

[FHSL96] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[FPSS96] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. The KDD process of extracting useful knowledge from volumes of data. *Communications of the ACM*, 39(11):27–34, November 1996.

[Fra94] J. Frank. Artificial intelligence and intrusion detection: Current and future directions. In *Proceedings of the 17th National Computer Security Conference*, October 1994.

[HLMS90] R. Heady, G. Luger, A. Maccabe, and M. Servilla. The architecture of a network level intrusion detection system. Technical report, Computer Science Department, University of New Mexico, August 1990.

[IKP95] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, March 1995.

[JLM89] V. Jacobson, C. Leres, and S. McCanne. tcpdump. available via anonymous ftp to ftp.ee.lbl.gov, June 1989.

[KFL94] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, pages 134–144, December 1994.

[KS95] S. Kumar and E. H. Spafford. A software architecture to support misuse intrusion detection. In *Proceedings of the 18th National Information Security Conference*, pages 194–204, 1995.

[KSSW97] J. O. Kephart, G. B. Sorkin, M. Swimmer, and S. R. White. Blueprint for a computer immune system. Technical report, IBM T. J. Watson Research Center, Yorktown Heights, New York, 1997.

[LB97] T. Lane and C. E. Brodley. Sequence matching and learning in anomaly detection for computer security. In *AAAI Workshop: AI Approaches to Fraud Detection and Risk Management*, pages 43–49. AAAI Press, July 1997.

[LSC97] W. Lee, S. J. Stolfo, and P. K. Chan. Learning patterns from unix process execution traces for intrusion detection. In *AAAI Workshop: AI Approaches to Fraud Detection and Risk Management*, pages 50–56. AAAI Press, July 1997.

[LTG$^+$92] T. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P. Neumann, H. Javitz, A. Valdes, and T. Garvey. A real-time intrusion detection expert system (IDES) - final technical report. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, February 1992.

[MTV95] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proceedings of the 1st International Conference on Knowledge Discovery in Databases and Data Mining*, Montreal, Canada, August 1995.

[Pax97] Vern Paxon. End-to-end internet packet dynamics. In *Proceedings of SIGCOMM '97*, September 1997.

[Pax98] Vern Paxon. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, 1998.

[PV98] Phillip A. Porras and Alfonso Valdes. Live traffic analysis of tcp/ip gateways. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, March 1998.

[SA95] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of the 21st VLDB Conference*, Zurich, Switzerland, 1995.

[SPT$^+$97] S. J. Stolfo, A. L. Prodromidis, S. Tselepis, W. Lee, D. W. Fan, and P. K. Chan.

Jam: Java agents for meta-learning over distributed databases. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*, pages 74–81, Newport Beach, CA, August 1997. AAAI Press.

[Sri96] R. Srikant. *Fast Algorithms for Mining Association Rules and Sequential Patterns.* PhD thesis, University of Wisconsin - Madison, 1996.

[Ste84] W. R. Stevens. *TCP/IP Illustrated*, volume 1. Addison-Wesley Publishing Company, 1984.

Securing 'Classical IP over ATM Networks'

Carsten Benecke   Uwe Ellermann
DFN-FWL[*]
Firewall-Laboratory for High-Speed Networks
Fachbereich Informatik, Universität Hamburg
Vogt-Kölln-Str. 30
D-22527 Hamburg
Phone: +49-40-5494-2262
Fax: +49-40-5494-2241
{Benecke,Ellermann}@fwl.dfn.de

Abstract

This paper discusses some security issues of 'Classical IP over ATM' networks. After analyzing new threats to IP networks based on ATM, security mechanisms to protect these networks are introduced. The integration of firewalls into ATM networks requires additional considerations. We conclude that careful configuration of ATM switches and ATM services can provide some level of protection against spoofing and denial of service attacks. Our solutions are intended to be applied to current IP over ATM networks and do not require any changes to these protocols or additions to current switch capabilities.

1 Introduction

The trend towards ATM networks requires a reexamination of network security issues. ATM is based on the concepts of switched virtual connections and fixed length cells, this contrasts with the connectionless, shared medium, broadcast networks frequently referred to as "legacy networks". These conceptual differences required the development of new protocols like 'Integrated Local Management Interface' (ILMI) [14] and 'Private Network-Network Interface' (P-NNI) [15]. These specifications have not yet been subjected to a thorough security analysis.

In order to make the use of IP in ATM networks, additional services, such as the ATMARP server[1], had to be introduced. This also introduced new risks, which must be investigated before "Classical IP over ATM networks" can be used in critical environments.

Typically cryptography is used in networks to provide authentication, integrity, and confidentiality. Integration of cryptographic mechanisms into ATM networks is currently a research topic [7, 16], but none of these mechanisms have been standardized.

We provide solutions for most identified security problems. Other security flaws can be mitigated. Many improvements are possible by manual configuration of ATM hardware and changes to the bahaviour of the ATMARP server. Thus there is no need to provide proprietary protocol extensions and security can be achieved within the current standards for IP over ATM. Moreover the solutions do not require additions to current switch capabilities like cryptographic authentication for signaling.

2 Attacks on "Classical IP over ATM Networks"

The model for the following security analysis of a "Classical IP over ATM" (CLIP) LAN consists of

[0] This work was funded by the DFN-Verein (Association for the promotion of a German Research Network) and Deutsche Telekom under project number: DT10.

[1] The ATMARP (ATM Address Resolution Protocol) as specified in [10] is required for resolving IP addresses into ATM addresses and vice versa. Unlike ARP [11] which uses broadcasts to resolve addresses a server is required in non broadcast multiple access networks such as ATM.

two logical IP subnets (LIS) connected to each other by a firewall. The firewall is used to divide the LAN into a critical subnet containing valuable data called the "internal network" (192.168.15.0) and a public subnet connected to the world called the "external network" (192.168.16.0). The TCP/IP traffic between the networks is examined by the firewall. The type of services provided and the access control policy enforced by the firewall will not be discussed here. Currently no "native ATM" applications need to be supported, but the concept must not prohibit extensibility.

One possible setup for the above requirements is shown in figure 1. This configuration has two major drawbacks. Firstly, it does not allow for the possibility of running a "native ATM" application using both of the networks, as the firewall only provides TCP/IP services. Secondly, as this configuration requires the use of two ATM switches, expensive equipment is wasted.

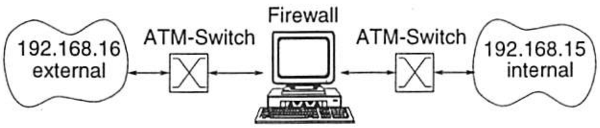


Figure 1: Gateway-Firewall in an ATM Network

As ATM allows for multiple *virtual networks* on the same physical network (i.e. on the same ATM switch) a similar setup can be built with just one ATM switch (see fig. 2).

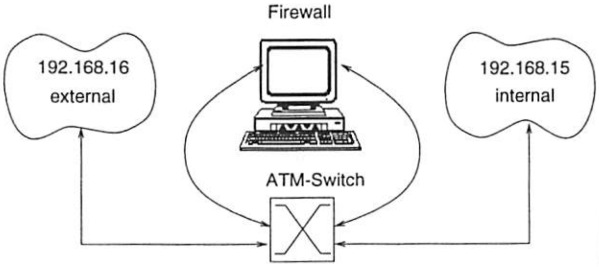


Figure 2: Gateway Firewall with one ATM Switch

The configuration of multiple virtual networks using one ATM switch is a simple task, but configuring this setup in a secure mode requires a deep analysis of the associated risks. The configuration with one ATM switch allows "native ATM" traffic to bypass the TCP/IP firewall. Unfortunately, "native ATM" connections can also be used to send IP datagrams. By circumventing the local Classical IP stack an attacker can send IP datagrams to systems in the other virtual network without traversing the firewall. The associated security risks and possible solutions are discussed in the following sections.

2.1 IP Spoofing over ATM Connections

IP spoofing attacks have been well understood for many years [2]. Essentially, IP spoofing means sending IP packets with faked IP source addresses. Services that use IP source addresses for authentication can be easily exploited by this attack.

IP spoofing is possible in 'Classical IP over ATM' (CLIP) networks. Whenever the ATM address of a server is known, an attacker can establish a direct ATM connection[2] to that host. The attacker can now register with the IP address of a trusted host by sending a carefully crafted 'InATMARP-Reply' message over this connection. After successful registration, spoofed IP packets can be sent over this connection. Moreover, due to the "ATMARP-Cache poisoning", the attacked server will send reply packets back to the attacker on the same ATM connection.

This kind of attack is possible, because the peers do not authenticate each other in a reliable manner. Moreover section *6.4 ATMARP Client Operational Requirements* of RFC 1577 [10] explicitly requires, that CLIP clients process 'InATMARP-Requests' and 'InATMARP-Replies' in order to update their local address resolution tables.

In contrast to legacy LANs (e.g. Ethernets) there is no need to attack the host whose address has been used[3]. Because the server sends its segments over the virtual connection to the attacker, the trusted host will not notice that its address has been used by another system.

Furthermore there is no need for the attacker to guess the TCP sequence number of the server. The server will use the established virtual connection to

[2] Note that the number of intermediate switches is irrelevant as long as a virtual connection between attacker and server can be established.

[3] In the case of a routed broadcast LAN the attacker also has to make sure that the host, whose IP address the attacker uses for spoofing, will not reset the spoofed connection. This can be done by flooding it with communication prior to the spoofing attack, so that the client is too busy to respond to the packets from the server.

send its segments to the attacker. So all other packets destined to the trusted host will also be sent to the attacker (see also section 2.3).

In summary, IP spoofing is easier to accomplish in ATM based networks and harder to detect. It can also be considered more dangerous, as simple countermeasures, for example requiring an IDENT [9] query before access is granted, can be spoofed more easily than in "legacy networks".

The figure 3 shows the time-sequence diagram of a successful simulated attack in a test environment. The attacker (A) pretends to be the host (192.168.15.A) and registers itself by sending an 'InATMARP-Reply' to host (B). In order to verify that the spoofing is successful, an 'ICMP-ECHO-Request' is used to test the established connection.

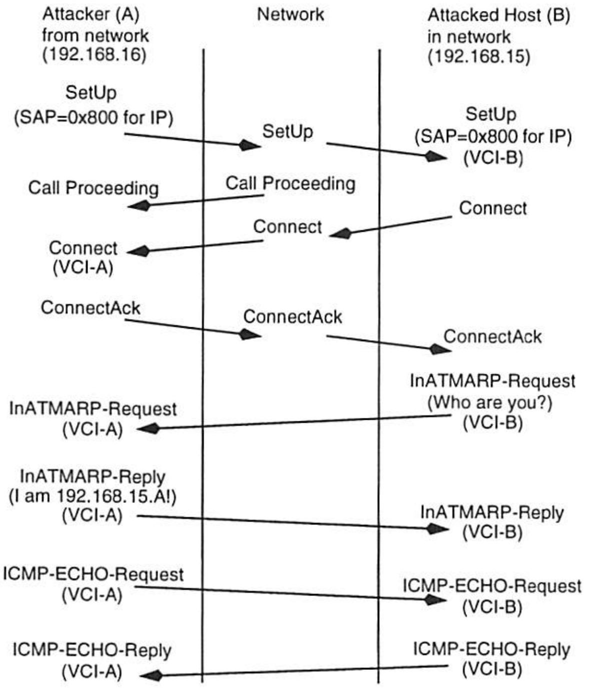


Figure 3: ATM based Spoofing Attack

It should be noticed that the attack is not detectable unless the host (B) verifies the 'InATMARP-Reply' by contacting a trusted ATMARP server (see section 3.1 for a discussion on securing the ATMARP server).

This attack is not restricted to hosts in the same LIS. Any host to which a direct ATM connection can be opened can be attacked. Routers that connect different LISs can be bypassed if the underlying ATM network allows for a direct ATM connection between hosts in different LISs. It is therefore important to point out that filters in these routers cannot be used to protect against this type of IP spoofing attacks. Neither a firewall nor an inter LIS router will have access to the datagrams because all hosts of the same LIS always use direct (non routed) connections. Moreover if the attacked host subsequentially wants to open TCP connections to the host (e.g. an IDENT query [9]), the address of which has been used for spoofing, a typical implementation will use the established virtual connection to the attacker. This would also enable the attacker to perpetrate 'Man in the Middle' attacks.

2.2 'Denial of Service' by Allocating IP Addresses of a LIS

The knowledge of the ATM address of an ATMARP server enables an attacker to scan the whole LIS IP address range. The attacker establishes a virtual connection to the ATMARP server and queries all IP addresses of interest. The server either replies with *'ATMARP_NAK'* or with the corresponding <*IP address,ATM address*> binding, providing all information an attacker needs for the denial of service attacks described below.

The attacker may now register itself with any unused IP address of the LIS. As every IP address can only be registered once, this will prevent hosts that were temporarily offline from registering themselves, if the attacker succeeds in allocating their IP address. The LIS member will be unable to use CLIP services as long as its IP address is registered with the attacker. Moreover the attacker may wait for clients to go offline by periodically verifying if a learned binding is still in the ATMARP servers cache.

Instead of reserving a random IP address in the subnet, the attacker only reserves the addresses of those LIS clients which are known to be temporarily offline. This puts the attacker in the position of being able to perpetrate very efficient attacks. Some additional aspects of ATMARP server based denial of service attacks are discussed in [1].

This kind of attack is possible, because any host that is able to establish an ATM connection to the ATMARP server, may register various <*IP address,ATM address*> bindings without authentication. An attacker only needs the ATM address of an ATMARP server in order to establish a virtual connection. Since the ATM address of the attacker is not used for path finding during signaling, the attacker may select any ATM address during connection establishment[4] with the ATMARP server. This makes it even harder to trace the origin of an attack, as the address information in the ATMARP server cannot be trusted.

2.3 'Man in the Middle' Attacks

Whenever a LIS member queries the ATMARP server for the corresponding ATM address of an IP address that has been allocated by an attacker, the ATMARP server will reply with an <*IP address,ATM address*> binding which has been supplied by the attacker. This enables the attacker to carry out various 'Man in the Middle' attacks, since the LIS member will connect to the attacker if the attacker has supplied his own[5] ATM address.

2.4 'Denial of Service' due to Bandwidth Allocation

Another problem arises from "native ATM" applications. These applications can use ATM specific services, e.g. allocating a constant bit rate (CBR) for their virtual connections. Since CLIP usually uses the 'best effort' service of ATM (unspecified bit rate connections) any CBR based communication has higher priority than CLIP traffic. These "native ATM" applications could accidentaly allocate the whole bandwidth of an intermediate switch. This results in an denial of service for IP based applications.

If an attacker uses bandwidth reservation, it suffices to signal the desired CBR rate. After establishing the connection there is no need to send any data over the virtual channel. All intermediate switches have agreed to the bandwidth allocation, so they cannot offer this bandwidth to other connections. From the attacker's point of view, this attack is very inexpensive, as there is no need to send (dummy) data with the allocated rate. Other traffic classes, such as variable bit rate (VBR), can also be used for this attack.

Moreover this kind of attack is hard to trace. The reservation of resources is a common procedure in ATM networks so that each intermediate switch may have to refuse the establishment of a connection due to insufficient resources. If a switch refuses the establishment of a CBR or VBR CLIP connection, the client host cannot detect whether this is due to normal protocol processing or vicious bandwidth allocation. There is also no way to deduce an attack by bad performance over an unspecified bit rate (UBR) connection since the throughput of an UBR connection may degrade at any time due to normal resource allocation in an ATM network.

2.5 'Denial of Service' due to Excessive Connection Establishment

Another kind of ATM based denial of service attack reserves all available virtual connection identifiers of CLIP hosts by establishing many connections to one machine. If this host tries to allocate an identifier for a CLIP connection there may be none left. Again this attack is ATM specific, because there is no way to send any datagrams unless an ATM connection has been established to the destination or a router. If this connection cannot be established due to insufficient virtual connection identifiers (VCID) it is not possible to access the desired CLIP service. The UNI 3.1 interface limits the maximum number of connections[6] to 2^{24}. It seems that this huge number of simultaneous connections is enough even for a big server but a typical client implementation of an ATM device driver limits the number of VCIDs to a considerable smaller number of 1024 or 2048. This limit of 2048 connections can be reached by an attacker trying to block a host from communication.

[4] RFC 1577 [10] section 5 "Overview of Call Establishment Message Content" requires the originator to supply a "Calling Party Number" Information Element (IE). It is expected to be an ATM address that really belongs to the calling system, but of course this IE can be faked like any other unauthenticated information.

[5] This will not necessarily identify the attacker's host because he may have registered an additional ATM address at his local switch (see also section 2.6).

[6] The ATM cells at the 'User to Network Interface' have 8 bits for virtual path identification and 16 bits for virtual channels. This allows for a theoretical total of 2^{24} different virtual connections at any time between host and switch.

2.6 ILMI based Attacks

The 'Integrated Local Management Interface' (ILMI, [14]) is used at the interface between switch and workstation. The protocol is based on the 'Simple Network Management Protocol' (SNMP). Whenever a workstation boots, it may automatically configure its ATM address by contacting the switch with ILMI messages. The switch usually supplies a 13 octet address prefix which is common to all hosts that are connected to the switch.

A problem arises by the fact that ILMI does not provide a mechanism for the authentication because the underlaying SNMP only uses clear text pass phrases[7] as a weak authentication mechanism. Moreover the community name to access the ILMI management information base (MIB) is specified as 'ILMI' [13, p. 147]. An attacker who does not need to authenticate himself can use the ILMI to register additional ATM addresses for his workstation. By using the additional registered address the attacker can bypass address filters which have been configured at the switch. The attacker could also try to register himself with the ATM address of an offline workstation. This is similar to the attack described in section 2.2. ILMI can also be used to automatically configure the interface type of an ATM switchport. An attacker may use ILMI to pretend that he is a switch by setting the interface type to "NNI" (Network to Network Interface).

In order to attack the switch the UNI signaling has to be changed to NNI[8] signaling. This has to be done prior to attacks on the switch to make sure, that the switch will recognize the P-NNI protocol as this will be used by the attacker.

Example: Simulating an ATM Switch

We will briefly discuss a possible strategy an attacker could use in order to prepare for the attack on a switch (see the following section). In typical "Plug and Play" installations it is likely that the attacker's host is connected to an ATM switch that uses the ILMI protocol for automatic configuration. The host has already been detected by the switch and the interface (port) of the switch is configured for UNI[9] signaling. In order to change the interface from UNI to NNI it is sufficient to use the following ILMI mechanisms:

- Send a *cold start trap* (SNMP) message to the switch. The switch will now assume that the peer interface management entity (IME) is reinitialized. The switch will clear all previously cached MIB information for this IME.
- Perform the ILMI connectivity procedures. The peer IMEs convince each other that they are connected.
- Perform the ILMI automatic configuration procedures. The switch will try to determine the type of the peer IME by querying the following MIB objects:
  - *atmfAtmLayerDeviceType* object. The attacker could answer with a value of 2, thus pretending to be a network node.
  - *atmfAtmLayerNniSigVersion* object. The attacker could answer with a value of 3, thus pretending to use the P-NNI routing protocol.

  The switch will now assume a symmetric[10] setup, and the attacker can use the P-NNI routing protocol to confuse the switch as described below.

2.7 Attacks on ATM Switches

The manipulation of a switch in an ATM network is very similar to attacking a router in a "legacy network" and presents a serious problem. An attacker might use the P-NNI protocol in order to manipulate a switch. He could inject incorrect information in the peer group[11] database or even try to configure routing loops into the hierarchic structure. He might block the communication of whole peer groups or even redirect communication over his

[7] RFC1157[3] denotes them as 'community names'.

[8] 'Network to Network Interface' (NNI) describes the appropriate interface for switch to switch interconnection.

[9] 'User Network Interface' (UNI) describes a protocol to be used for connection management between host and private ATM switches.

[10] If the P-NNI protocol is used at the NNI, the setup is called *"symmetric"* because there are two network nodes (switches). The UNI protocols are not symmetric because they are used for different kinds of peers (between an end system (host) and a network node (switch)).

[11] A number of switches that share a common addressing scheme, e.g. the same address prefix, are grouped together. They belong to a 'peer group'.

workstation. The blocking of a peer group is very similar to the manipulation of routers with incorrect 'ICMP-Host/Net- unreachable' messages.

Attacks based on the P-NNI protocol can use replies to 'HELLO' messages of a peer group leader to inject malicious information about 'link states'. The peer group leader in turn will broadcast these changes to its group members. Peer group members that have updated their link state information with faked information are likely to make the wrong routing decision.

3 Solutions

Attacks on the ARP service and the ATM based IP spoofing can be prevented, if the ATM switch or, more generally, the ATM network supports mechanisms for access control. Most other discussed attacks can be prevented or mitigated by careful configuration of the ATM switch.

In ATM networks the control on access[12] to the network is shared between hosts and switches. On the other hand in "legacy LANs" like Ethernet the access to the network is only controlled by the nodes themselves (CSMA/CD)[13]. This fundamental difference between ATM and "legacy LANs" can be used for securing the networks against malicious attacks by providing a controlled access to network resources.

A major part of securing an ATM network relies on the ATM switch. It can be used for filtering certain addresses and to support monitoring of the network. The next sections will show how an ATM network can be secured against the attacks described before.

3.1 Securing the ATMARP Service

CLIP requires one ATMARP server for each LIS [10]. The ATM address of this server has to be configured on every node in the same LIS. The protection of the ATMARP service requires three steps:

- Configuration of the ATMARP server on one node in the LIS
- Configuration of static ARP entries for each known node in the LIS
- Auditing of all ATMARP queries coming from hosts that were not configured with a static ARP entry.

Configuration of an ATMARP Server An ATMARP server can be installed on a host in the LIS or on an ATM switch. From the security point of view installing an ATMARP server on a host is preferable for multiple reasons. The access to the ATMARP server on a host can be controlled more easily than on an ATM switch. As an ATMARP server on a host is implemented in software, it is easier to expand the ATMARP server with security enhancements for access control and auditing. ATMARP server on ATM switches depend on the firmware of the switch and are therefore difficult to expand. Moreover even though it is convenient to use the ATMARP server of the switch for multiple LISs, this makes it impossible to restrict the access according to the security policy of one LIS.

Configuring static ARP Entries In the second step the ATMARP server is configured with static ARP entries for every node in the LIS. Each entry consists of the tuple *<ATM address, IP address>*. This renders the described attack of malicious registration of an address of another node impossible, as static entries cannot be overwritten.

To gain additional security the ATMARP server should be configured to reject registration requests for unknown tuples of *<ATM address, IP address>*. This can be achieved by turning off the dynamic learning of new *<ATM address, IP address>* tuples. By these simple measures hostile registration of ARP entries can be prevented.

It has to be noted however that adding new nodes to this network will require manual configuration of the ARP entry.

Auditing of ATMARP Queries Attempts to register new ARP entries can be identified as either attacks or newly installed machines and should be audited. Also queries for unregistered ARP entries

[12] During signaling for connection establishment any node (both peers) and any intermediate switch may disagree to the SETUP request. ATM networks therefore offer some kind of "shared control" in contrast to legacy LANs which usually offer only a "shared access".

[13] Carrier Sense Multiple Access with Collision Detection

should be audited as this can be an attempt to scan the address-space for possible victims.

Open Issue In RFC1577 [10] it is required that every LIS client opens an ATM connection to the ATMARP server and registers its address binding. The ATMARP server will reject the registration if the same binding is already registered for another ATM connection. This attack will also succeed for statically configured <*ATM address, IP address*> bindings, as the ATMARP server will accept the malicious registration because a valid binding has been supplied. In consequence even with statically configured bindings a client may be blocked from registering after reboot. A solution to this problem is provided in the next section.

3.2 Integration of the Switch into a Firewall Concept

As described before, an attacker could try to open an ATM connection from an external host to an internal host thereby circumventing a firewall. A firewall concept for an ATM network needs to integrate the switch to prevent these attacks from succeeding. The configuration of the ATM switch has to make sure that connections can only be established between hosts of the same subnetwork (external or internal network). This can be done either by configuring PVCs only[14] or by restricting the signaling of new SVCs with access control list. The configuration of PVCs between all hosts of the same subnetwork yields a very secure configuration, but is hard to administer as adding a host will require the configuration of PVCs to *every* host in the same subnet. The definition of access lists on the other hand is more flexible and efficient as only few simple rules can restrict the creation of new SVCs. Even adding and removing hosts may not require reconfiguration if the subnets are identified in the access control lists by the use of wildcards.

The following sections describes the secure separation of two subnets – an internal and an external subnet – using access control lists on the signaling protocol. The firewall has two ATM interfaces: one "fwext" with the external network and one "fwint" with the internal network (see fig. 4).

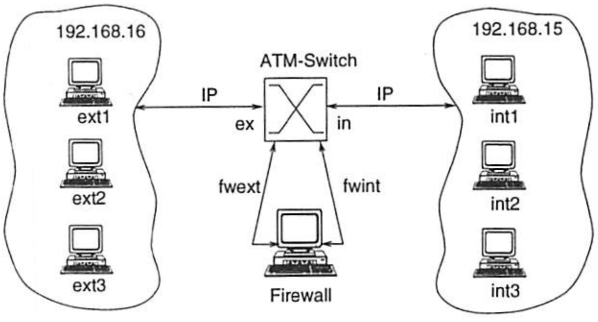


Figure 4: Firewall in an ATM Environment

3.2.1 Access Control on Signaling

Switches take an active part in opening ATM connections. Therefore ATM switches can be used to increase security by restricting the hosts ability to open connections to other hosts. The filters are similar to the rules that can be defined in packet screens. But these two filters differ in two important aspects: packet screens filter every packet whereas ATM switches restrict the creation of new ATM connections — no filtering occurs after successful setup of a connection. Rules in packet screens can be formulated on hosts (IP addresses) and services (TCP/UDP ports) whereas ATM switches only have access to the ATM addresses of the communicating hosts. The ATM switches have no control over type of IP-based service that will be accessed by the hosts. The filters on ATM addresses are sufficient for the intended use, as hosts will be grouped together by their ATM addresses and a firewall will be used for additional filters depending on the type of service.

As the filters on the switch rely on ATM addresses only, both the internal and the external network may have complex topologies (e.g. multiple interconnected switches). It is sufficient to configure the filters on the switch that is connected to the firewall. Figure 4 shows a simplified setup with only one switch.

The following restrictions have to be imposed by the ATM switch:

- Connections between hosts on the internal network should not be restricted.
- No connections should be opened from an internal to an external host.

[14] Beside configuring the PVCs the signaling of SVCs must be disabled.

- No connections should be opened from an external to an internal host.
- Connections between hosts on the external network should not be restricted.

With this configuration all communication between the internal and the external network must cross the firewall.

3.2.2 Sample Configuration for a Cisco LS 1010 Switch

The advantages of filter rules in ATM switches are described with the following example. The example network consist of three hosts on the internal network ("int1-3"), three hosts on the external network ("ext1-3"), one firewall with two interfaces ("fwint" and "fwext") and one ATM switch ("atmsw").

Definition of Symbolic Names for ATM Addresses Symbolic names are easier to comprehend and therefore reduce the risks of misconfigurations:

```
atm template-alias fwint
        4700918100000000e0f7df1901080020827e6100
atm template-alias fwext
        4700918100000000e0f7df1901080020827c5100
atm template-alias int1
        4700918100000000e0f7df1901080020827b8100
atm template-alias int2
        4700918100000000e0f7df1901080020825ca100
atm template-alias int3
        4700918100000000e0f7df190108002082a9f100
atm template-alias atmsw
        4700918100000000e0f7df1901f9f9f9f9f9f900
atm template-alias ext1
        4700918100000000e0f7df1901080020827be100
atm template-alias ext2
        4700918100000000e0f7df19010800208256410
atm template-alias ext3
        4700918100000000e0f7df19010800208254d100
```

In this example we choose to explicitly list all hosts in the filter rules. For larger networks wildcards should be used to select groups of hosts with just one rule.

Definition of Simple Access Control Lists The keywords `permit` and `deny` can be used to describe which hosts are allowed to use the signaling protocol to open a connection:

```
# internal hosts are allowed to reach
#                 other internal hosts
atm filter-set inHosts permit fwint
atm filter-set inHosts permit int1
atm filter-set inHosts permit int2
atm filter-set inHosts permit int3
atm filter-set inHosts permit atmsw
atm filter-set inHosts deny default

# Hosts not listed (external hosts)
# are not allowed to reach internal hosts
atm filter-set exHosts deny   fwint
atm filter-set exHosts deny   int1
atm filter-set exHosts deny   int2
atm filter-set exHosts deny   int3
atm filter-set exHosts deny   atmsw
atm filter-set exHosts permit default

atm filter-exp intern inHosts
atm filter-exp extern exHosts
```

Unknown hosts will be treated like external hosts and cannot reach internal hosts. As only internal ATM addresses are necessary for the configuration the solution is applicable to both LANs and WANs.

If the ATMARP server has been configured as internal host, the address filters are sufficient to prevent an external attacker from directly connecting to the ATMARP server (see section 3.1). The combination of a secure ATMARP server and the use of address filters on switches to direct the access to the network through a firewall is a good choice for defending against the denial of service attacks discussed in sections 2.2 and 2.5.

Applying Filters to Interfaces The filters must be applied to the interfaces[15]. On an interface connected to an internal host the following commands must be applied:

```
atm access-group intern in
atm access-group intern out
```

On all external interfaces these two commands must be applied:

[15] Filters are not in use unless they are applied to a port of the switch.

```
atm access-group extern in
atm access-group extern out
```

3.2.3 Extension of the Concept

The above configuration separates the ATM network into two virtual networks the internal and the external network. This concept can be extended to separate multiple virtual networks. To which network a host belongs is simply a matter of configuration. A host may therefore belong to a virtual network depending on its communication and security requirements rather than depending on its physical location.

This feature of ATM networks makes ıt easier to structure networks depending on security requirements. If all these subnets are connected by one or multiple central firewalls, this concept will allow for very secure networks in the future.

3.3 Mitigation of Bandwidth Allocation Attacks

The combination of firewall, access control lists for signaling on the switch, and the secure ATMARP server is sufficient to establish a secure 'Classical IP over ATM' network. Figure 5 shows an extension to the setup discussed in section 4. The switch used for access control to the secure network is also used to connect other LISs ("external") to the Internet. As there is only one physical wire to the Internet access point (connected to the "ex" interface of the switch) it is still possible that the available bandwidth over this wire is allocated by an attacker (visualized with a dashed line between H1 and H2). The internal hosts will not be able to connect to the Internet or will perform at least very poorly.

There are two solutions that can address this problem. Both solutions require a careful configuration of the switch.

3.3.1 Limiting the available Bandwidth for each Traffic Class

Modern switches support the configuration of bandwidth resources for different traffic classes. Thus it may be possible to configure the "fwext" interface of the switch (figure 5) so that only a certain amount of bandwidth is available for CBR or VBR traffic. In essence this will not prevent the described attack but will reduce the impact, as the administrator can make sure that there is always enough bandwidth available for connections to the Internet.

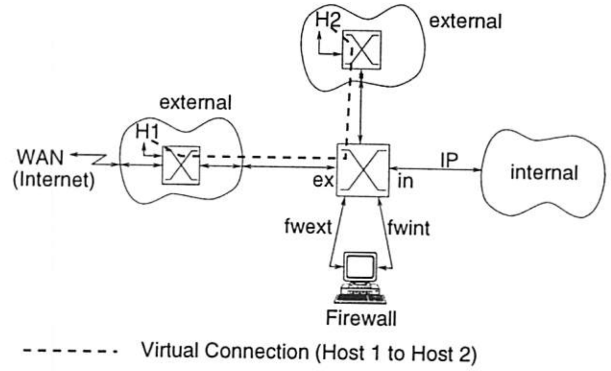


Figure 5: Bandwidth Allocation for SVC between two External Hosts

3.3.2 Configure a Permanent Virtual Connection

Another way to make sure that the internal network is always able to connect to the Internet is to configure a permanent virtual connection between switch and the Internet access point (router or switch). Typical switches provide a way to assign quality of service to PVCs. This is sufficient to allocate a certain amount of bandwidth to the PVC so that an attacker can only compete for the remaining bandwidth.

The main drawback of both solutions is that the administrator must have access to all intermediate switches in order to configure either a PVC to the Internet access point or limit the available bandwidth for demanding traffic classes (CBR/VBR). This may be impossible if the switches belong to different administration domains. Moreover the access to all the intermediate switches must be restricted to prevent reconfiguration by an attacker.

3.4 Static Switch Configuration

Both the ILMI based attacks and the attacks on the switch (see section 2.6 and 2.7) can be prevented by static switch configuration. For example the administrator may configure individual ports of a switch as UNI ports. This prohibits changes to the port settings by a peer as described in section 2.6. It should be noted that this is a local configuration that needs to be applied to one switch only. This configuration will prevent a workstation which is connected to a port of the switch to use the P-NNI protocol for attacks on the switch.

Since there is still the chance that malicious routing information is propagated by other switches over a NNI link the switch is still vulnerable. One (interim) solution to this problem is the use of the 'Interim Inter-switch Signaling Protocol' (IISP) instead of P-NNI at the NNI interface unless the authentication has been addressed by the P-NNI designers. The IISP only supports statically configured routes, thus making it impossible to insert malicious information by sending messages to the switch. The major drawback of this solution is the burden of managing the static routing tables for fast scaling networks. IISP networks are also very error prone. If an intermediate switch goes offline all static routes through this switch are no longer usable.

A switch may also support the configuration of static ATM addresses at individual ports. This can prevent the registration of additional (unexpected) addresses at a switch port. This also makes sure that a workstation will only be able to register with a predefined address that matches a desired filter rule on the switch (see section 3.2).

4 Conclusions

This article describes some vulnerabilities present in "Classical IP over ATM" networks and introduces a switch based configuration and extensions to the ATMARP service as a countermeasure.

The use of IP in ATM networks leads to some interesting security problems. The risks of IP spoofing attacks are still high in ATM networks and need to be addressed by appropriate security mechanisms. In addition to these well known risks ATM features some new protocols whose security implications are not yet fully understood. Some possible attacks based on ILMI and P-NNI have been introduced in sections 2.6 and 2.7.

Section 3.1 discusses methods on how to secure an ATMARP server. ATMARP is a critical service for CLIP networks that must be secured. Another important result is that the ATM switches are very important for securing 'Classical IP over ATM' networks. ATM offers a "shared control" to network resources (see section 3). This feature is the basis for access control mechanisms in ATM switches (section 3.2.1). As an example we have shown how to setup ATM address filters for guarding the access to secure networks.

As ATM address filters are not sufficient to enforce typical security policies, firewalls will have to be used in combination with ATM address filters. The integration of a firewall into an ATM network was discussed for a gateway firewall. The concept can easily be expanded for a combination of packet screen with bastion host. This would require the configuration of three subnets (internal, external and DMZ[16]) instead of two subnets (internal and external). The switch can enforce the separation of these three virtual subnets with the same mechanisms that have been described before.

Section 3.3 shows how switches can be used to prevent some denial of service attacks by static configuration. This is necessary unless the ATM based protocols such as P-NNI and ILMI offer some kind of authentication.

Many of the problems discussed here originate from "Plug and Play" configurations. Vendors tend to supply their switches with automatic configuration tools (such as ILMI) which enable easy network setups. But a secure network requires a careful manual configuration of switches, protocols, and devices (e.g. firewalls) that control access to the network.

References

[1] Armintage, G. 1997: "Security Issues for ION protocols", Internet-Draft `draft-armitage-ion-security-00`, work in progress, Lucent Technologies 1997

[2] Bellovin, S.M. 1989: "Security Problems in the TCP/IP Protocol Suite" in Computer Communication Review, Vol. 19, No. 2, pp. 32–48, April 1989

[16] A bastion host is usually installed on its own subnet, frequently called "Demilitarized Zone" (DMZ) [4, 5].

[3] Case, J. et al 1990: "Request for Comments 1157: A Simple Network Management Protocol (SNMP)", Network Working Group, May 1990

[4] Chapman, D. B.; Zwicky, E. D. 1995: "Building Internet Firewalls", O'Reilly & Associates, September 1995.

[5] Cheswick, W. R.; Bellovin, S. M. 1994: "Firewalls and Internet Security: Repelling the wily Hacker", Addison-Wesley, 1994.

[6] Danthine, A.; Bonaventure, O. 1995: "Is ATM a Continuity or a Discontinuity for the LAN Environment?" in Effelsberg, W.; Spaniol, O.; Danthine, A. (eds.) Proceedings: HSN for Multimedia Applications, Dagstuhl, June 1995

[7] Deng, H.; Gong, L.; Lazar, A. 1995: "Secure Data Transfer in Asynchronous Transfer Mode Networks". *In: Proceedings of IEEE Globecom '95, Singapore*, November 1995.

[8] Heinanen, J. 1993: "Request for Comments 1483: Multiprotocol Encapsulation over ATM Adaptation Layer 5", Network Working Group, July 1993

[9] Johns, M. St. 1993: "Request for Comments 1413: Identification Protocol", Network Working Group, February 1993

[10] Laubach, M. 1994: "Request for Comments 1577: Classical IP and ARP over ATM", Network Working Group, January 1994

[11] Plummer, D. C. 1982: "Request For Comments 826: An Ethernet Address Resolution Protocol", Network Working Group, November 1982

[12] ATM Forum 1994: "Interim Inter-switch Signaling Protocol (IISP) Specification Version 1.0", ATM Forum 1994

[13] ATM Forum 1994: "ATM User-Network Interface (UNI) Specification Version 3.1", ATM Forum 1994

[14] ATM Forum 1996: "Integrated Local Management Interface (ILMI) Specification Version 4.0", ATM Forum 1996

[15] ATM Forum 1996: "Private Network-Network Interface Specification Version 1.0", ATM Forum 1996

[16] Varadharajan, V; Shankaran, R.; Hitchens, M. 1997: "Security Issues in Asynchronous Transfer Mode". *Proceedings of Second Australasian Conference, ACISP'97, Sydney July 1997*, Springer, 1997.

A Java Beans Component Architecture for Cryptographic Protocols

Pekka Nikander
pekka.nikander@hut.fi
Helsinki University of Technology

Arto Karila
arto.karila@hut.fi
Helsinki University of Technology

Abstract

Global networking has brought with it both new opportunities and new security threats on a worldwide scale. Since the Internet is inherently insecure, secure cryptographic protocols and a public key infrastructure are needed. In this paper we introduce a protocol component architecture that is well suited for the implementation of telecommunications protocols in general and cryptographic protocols in particular. Our implementation framework is based on the Java programming language and the Conduits+ protocol framework. It complies with the Beans architecture and security API of JDK 1.1, allowing its users to implement application specific secure protocols with relative ease. Furthermore, these protocols can be safely downloaded through the Internet and run on virtually any workstation equipped with a Java capable browser[*]*. The framework has been implemented and tested in practice with a variety of cryptographic protocols. The framework is relatively independent of the actual cryptosystems used and relies on the Java 1.1 public key security API. Future work will include Java 1.2 support, and utilization of a graphical Beans editor to further ease the work of the protocol composer.*

1 Introduction

Designing and implementing telecommunications protocols has proven to be a very demanding task. Building secure cryptographic protocols is even harder, because in this case we have to be prepared for not just random errors in the network and end-systems but also premeditated attackers trying to take advantage of any weaknesses in the design or implementation [3] [29]. During the last ten years or so, much attention has been focused on the formal modelling and verification of cryptographic protocols [21] [27]. However, the question how to apply these results to real design and implementation has received considerably less attention [17]. Recent results in the area of formalizing architecture level software composition and integrating it with object oriented modelling and design seem to bridge one section of the gap between the formal theory and everyday practice [2] [16] [31].

In this paper we present a practical architecture and an implementation framework for building secure communications protocols that have the following properties:

- The architecture is made to the needs of today's applications based on the global infrastructure that is already forming (Internet, WWW, Java).
- The implementation framework allows us to construct systems out of our own trusted protocol components and others taken from the network. These systems can be securely executed in a "protocol sand box", where they, for example, cannot leak encryption keys or other secret information.
- Together they allow us to relatively easily implement application specific secure protocols, securely download the protocol software over the Internet and use it without any prior arrangements or software installation.

We have implemented the main parts of the architecture as an object oriented protocol component framework called Java Conduits. It was built using JDK 1.1 and is currently being tested on the Solaris operating system. The framework itself is pure Java and runs on any Java 1.1 compatible virtual machine.

Our goal is to provide a sound practical basis for protocol development, with the desire to create higher level design patterns and architectural styles that could be formally combined with protocol modelling and analysis. The current focus lies in utilizing the "gang of four" object level design patterns [10] to create a highly stylistic way of building both cryptographic and non-cryptographic communications protocols. Our implementation experience has shown that this approach leads to a number of higher level design patterns, i.e., protocol

[*] In order to achieve real sandbox security, either JDK 1.2 or a specially tailored SecurityManager is needed [12].

patterns, that describe how protocols should be composed from lower level components in general.

The rest of this paper is organized as follows. In Section 2 we introduce our architecture and its relationship to existing work. In Section 3 we present the component framework developed. Section 4 dwells into implementational details and experience gained while building prototypes of real protocols. At the end we present a summary (Section 5) and outline some future work (Section 6).

2 The architecture

In our view, the world to which we are building applications consists of the following main components: the *Internet*, the *World Wide Web* (WWW), the *Java programming language and execution environment* and an *initial security context* (based on predefined trusted keys). Our architecture is based on these four corner stones. In addition, there are three more components that are not indispensable but "nice-to-have": a *Public Key Infrastructure* (PKI), the *Internet Security Association and Key Management Protocol* (ISAKMP) and the *Internet Protocol Security Architecture* (IPSEC).

2.1 The essential components

The world-wide Internet has established itself as the dominating network architecture that even the public switched telephone network has to adapt to. The new Internet Protocol IPv6 will solve the main problem of address space, and together with new techniques, such as resource reservation and IP switching, provide support for new types of applications, such as multimedia on a global scale. As we see it, the only significant threats to the Internet are political, not technical or economic. We regard the Internet, as well as the less open extranet and intranet, as an inherently untrusted network.

The World Wide Web (WWW) has been the fastest growing and most widely used application of the Internet. In fact, the WWW is an application platform which is increasingly being used as an user interface to a multitude of applications. Hyper Text Markup Language (HTML) forms and the Common Gateway Interface (CGI) make it possible to create simple applications with the WWW as the user interface. More recently, we have seen the proliferation of executable content.

The Java programming language extends the capabilities of the WWW by allowing us to download executable programs, Java applets, with WWW pages. A Java virtual machine has already become an essential part of a modern web browser and we see the proliferation of Java as being inevitable. We are basing our work on Java and the signed applets security feature of Java 1.1.

In order to communicate securely, we always need to start with an initial security context. In our architecture, the minimal initial security context contains the trusted keys of our web browser, which we can use to check the signatures of the downloaded applets and other Java Beans.

2.2 The optional components

While our architecture does not depend on the existence of the following three components, they are "nice to have", as they will make the architecture more efficient and scalable.

A public key infrastructure (PKI) allows us to associate a public key with a person, company, service, authorization, or such with a reasonable assurance level. It also allows us to prove the authenticity of a digital signature in a court of law. A global PKI is a prerequisite for many new application areas for the Internet. Until recently, most of the work in this area has focused on X.509 type certificates and a hierarchical tree of certification authorities (CAs). While this approach works for some application areas, e.g., in relations between governments, it is not suitable for others, since trust is inherently intransitive. The Simple Public Key Infrastructure (SPKI) [9] appears to us as a more widely applicable PKI.

The Internet Security Association and Key Management Protocol (ISAKMP) [19] provides us with a standard way of securely generating keys and setting up security contexts. We expect a number of application-specific security protocols to be built on top of ISAKMP. The authentication information needed for securing a connection can easily be augmented with capabilities such as authorization information. This allows future access control policies to be based on signed authority in addition to explicit identity.

The Internet Protocol Security Architecture (IPSEC) [6] [30] is an extension to IPv4 and an essential part of IPv6. It provides us with authenticated, integral and confidential channels for transparent exchange of information between any two hosts, users or programs on the Internet. Designed to be used everywhere, it will be implemented on most host and workstation operating systems in the near future. The flexible authentication schemes provided by ISAKMP make it possible to individually secure single TCP connections and UDP packet streams.

IPSEC is not yet ubiquitously available, so, for now, its functionality can be substituted with an transport layer protocol such as SSL. The current JDK architecture does not allow IPSEC to be implemented in Java without resorting to native interfaces that allow access to the underlying protocol stack or media.

2.3 Implementational requirements

Future protocols will be drastically different from what most of us believed only a few years ago. The role of security cannot be over-emphasized. Unfortunately, most of the tools and frameworks developed so far either tend to ignore security or do not facilitate integrating protocol security with that of the underlying operating system or the supported applications. This is unacceptable, since security should be designed and built in to the protocols and the system as a whole from the very beginning.

The earlier protocol frameworks were typically based on a virtual operating environment that was clearly separated from the underlying operating system. From the modularization point of view this was good. However, this made it hard to build application level programs that were able to use the protocols running within the framework. Java Conduits is clearly different in this respect. For example, under the JavaOS, the protocols and the applications all run within a single virtual environment, making seamless integration straightforward.

The use of an object oriented implementation language allows us to extensively use object-level design patterns. This makes the framework itself more generic and extensible, and creates a highly stylistic way for writing actual protocol implementations. With a suitable object oriented design tool, the outline for the classes needed to implement a new protocol can be created in minutes. The actual implementation code for the protocol actions typically takes a little longer, depending on the complexity of the protocol.

Performance will always be an issue with communications protocols. Even though processing power is constantly increasing, the new applications need ever-increasing bandwidth and reasonable transfer delay. The new protocols require large transfer capacity, short and fixed delay, and lots of cryptography, among other things.

There are two facets to performance. First, the processing power available should be used as efficiently as possible. The importance of this will gradually decrease as processing power increases. Second, and more important, there should not be any design limitations which set a theoretical limit to the performance of the protocols, no matter how much processing power we have. We want to allow as much parallelism as possible and build the protocol implementations such that they can be efficiently divided between a number of processors. Java, with its built-in threads and synchronization, allows parallelism to be utilized with relative ease.

2.4 Related work

Our implementation framework is heavily based on the ideas first presented with the x–Kernel [15] [18] [22] and the Conduits [32] and Conduits+ [14] frameworks. Some of the ideas, especially the microprotocol approach, have also been used in other frameworks, including Isis [8], Horus/Ensemble [24], and Bast [11]. However, Isis and Horus concentrate more on building efficient and reliable multiparty protocols, while Bast objects are larger than ours, yielding a white box oriented framework instead of a black box one.

Compared to x–Kernel, Isis and Horus, our main novelty is in the use and recognition of design patterns at various levels. Furthermore, our object model is more fine-grained. These properties come hand-in-hand — using design patterns tends to lead to collections of smaller, highly regular objects.

The Horus/Ensemble security architecture is based on Kerberos and Fortezza. Instead, we base our architecture on the Internet IPSEC architecture. Kerberos does not scale well and requires a lot of trusted functionality. Fortezza is developed mainly for U.S. Government use, and not expected to be generally available. On the other hand, we expect the IPSEC architecture to be ubiquitously available in the same way as the Domain Name System (DNS) is today.

Most important, our framework is seamlessly integrated into the Java security model. It utilizes both the language level security features (packages, visibility) and the new Java 1.1 security functionality. A further difference is facilitated by the Java run time model. Java supports code and object mobility. This allows application specific protocols to be loaded or used on demand.

Another novelty lies in the way we use the Java Beans architecture. This allows modern component based software tools to be used to compose protocols. The introduction of the Protocol class, or the metaconduit (see Section 3.2), which allows composed subgraphs to be used as components within larger protocols, is especially important. The approach also allows the resulting protocol stacks to be combined with applications.

3 The implementation framework

Java Conduits provides a fine grained object oriented protocol component framework. The supported way of building protocols is very patterned, on several levels. The framework itself utilizes heavily the "gang of four" object design patterns [10]. A number of higher level patterns for constructing individual protocols are emerging. At the highest level, we envision a number of architectural patterns to surface as users will be able to construct protocol stacks that are matched to application needs.

Our goal is to allow application-specific secure protocols to be built from components. The protocols themselves can be constructed from lower level components, called conduits. The protocol components, in turn, can be combined into complete protocol stacks. To achieve this, we have to solve a number of generic problems faced by component based software.

3.1 Component based software engineering

Recently, attention has shifted from basic object oriented (OO) paradigms and object oriented frameworks towards combining the benefits of OO design and programming with the broad scale architectural viewpoints [2] [20]. Component based software architectures and programming environments play a crucial role in this trend.

For a long time, it was assumed that object oriented programming alone would lead to software reusability. However, experience has shown this assumption false [20]. On the other hand, non object oriented software architectures, such as Microsoft OLE/COM and IBM/Apple OpenDoc, have shown modest success in creating real markets for reusable software components. Early industry response seems to indicate that the Java Beans architecture may prove more successful.

The Java Beans component model we are using defines the basic facets of component based software to be *components*, *containers* and *scripting*. That is, component based software consists of component objects that can be combined into larger components using containers. The interaction between the components can be controlled by scripts that should be easy to produce, allowing less sophisticated programmers and users to create them. This is achieved through *runtime interface discovery*, *event handling*, *object persistence*, and *application builder support*. [33]

Java as a language provides natural support for run–time interface discovery. A binary Java class file contains explicit information about the names, visibility and signatures of the class and its fields and methods. Originally provided to enable late loading and to ease the fragile superclass problem, the runtime environment also offers this information for other purposes, e.g., to application builders. Java 1.1 provides a reflection API as a standard facility, allowing any authorized class to dynamically find out and access the class information.

The Java Beans architecture introduced a new event model for Java 1.1. The model consists of *event listeners*, *event objects* and *event sources*. The mechanism is very lean, allowing basically any object to act as a event source, event listener, or even the event itself. Most of this is achieved through class and method naming conventions, with some extra support through manifestational interfaces.

Compared to other established component software architectures, i.e., OLE/COM, CORBA and OpenDoc, the Java Beans architecture is relatively light-weight. Under Java 1.1, nearly any object can be turned into a Java Bean. If an object's class supports serialization* and the object does not contain any references to its environment, the object can be considered to be a Bean without any changes at all. When Bean properties are provided by naming access functions appropriately, event support added with a few lines of code, and any references to the enclosing environment marked transient, almost any class can be easily turned into a Bean.

On the other hand, the Java Beans architecture, as it is currently defined, does not address some of the biggest problems of component based software architectures any better than its competitors. These include the mixing and matching problem that faces anyone trying to build larger units from the components. Basically, each component supports a number of interfaces. However, the semantics of these interfaces are often not immediately apparent, nor can they be formally specified within the component framework. When the components are specifically designed to co-operate, this is not a problem. However, if the user tries to combine components from different sources, the interfaces must be adapted. This may, in turn, yield constructs that cannot stand but collapse due to semantic mismatches.

In the protocol world, the mixing and matching problem is reflected in two distinct ways. First, the data transfer semantics differ. Second, and more importantly, the information content needed to address the intended recipient(s) of a message greatly differ. In our framework, the recipient information is always implicitly available in the topology of the conduit graph. Thus, the protocols have no need to explicitly address peers once an appropriate conduit stream has been created.

It has been shown that secure cryptographic protocols, when combined, may result in insecure protocols [13]. This problem cannot be easily addressed within the current Java Beans architecture. We hope that future research, paying more attention to the formal semantics, will alleviate this problem.

* A Java class supports serialization by manifesting implementation of the `java.lang.Serializable` interface. Most Java classes can do this. However, there are classes that are inherently impossible to be serialized as such, e.g., `java.lang.Thread`.

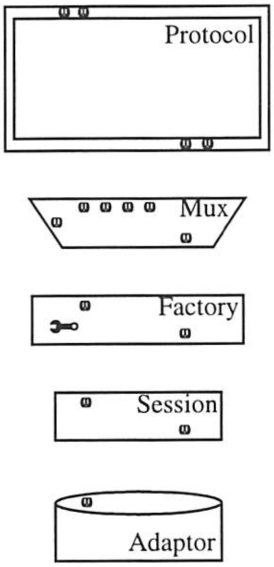


Figure 1: The five types of conduits

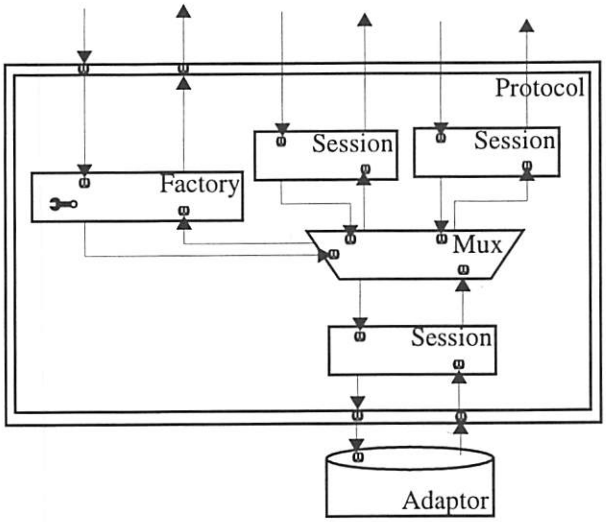


Figure 2: Aa example of a simple partial protocol graph

3.2 Basic Conduits architecture

The basic architecture of Java Conduits is based on that of Conduits+ by Hueni, Johnson and Engel [14]. The basic kinds of objects used are *conduits* and *messages*. Messages represent information that flows through a protocol stack. A conduit, on the other hand, is a software component representing some aspect of protocol functionality. To build an actual protocol, a number of conduits are connected into a graph. Protocols, moreover, are conduits themselves, and may be combined with other protocols and basic conduits into larger protocol graphs, representing protocol stacks.

There are five kinds of conduits: *Session, Mux, ConduitFactory, Adaptor* and *Protocol*. Each conduit has two sides: side A and side B. A given conduit can connect to either side A or side B of another conduit.

Sessions are the basic functional units of the framework. A session implements the finite state machine of a protocol, or some aspects of it. The session remembers the state of the communication and obtains timers and storage for partial messages from the framework. The session itself does not implement the behaviour of the protocol but delegates this to a number of State objects, using the State design pattern.

The Mux conduits are used to multiplex and demultiplex protocol messages. In practical terms, a Mux conduit has one side A that may be connected to any other conduit. The side B[0] of the Mux is typically connected to a ConduitFactory. In addition, the Mux has a number of additional side B[i] conduits. Protocol messages arriving from these conduits are multiplexed to the side A conduit, and vice versa.

If the Mux determines, during demultiplexing, that there is no suitable side B[i] conduit to which a message may be routed, the Mux routes the message to the ConduitFactory attached to side B[0]. The ConduitFactory creates a new Session (or Protocol) that will be able to handle the message, installs the newly created Session to the graph, and routes the message back to the Mux.

Adaptors are used to connect the conduit graph to the outside world. In conduit terms, adapters have only side A. The other side, side B, or the communication with the outside world, is beyond the scope of the framework, and can be implemented in whatever means feasible. For example, a conduit providing the TCP service may implement the Java socket abstraction.

A protocol is a kind of metaconduit that encapsulates several other conduits. A protocol has sides A and B. However, typically these are conduit connections that are mainly used for the delivery of various kinds of interprotocol control messages. Typically the actual data connections directly stretch between the conduits that are located inside some protocols. In practice, a protocol is little more than a conduit that happens to delegate its sides, i.e., side A and side B independently, to other conduits. The only complexity lies in the building of the initial conduit graph for the protocol. Once the graph is built, it is easy to frame it within a protocol object. The protocol object can then be used as a component in building other, more complex protocols or protocol stacks.

3.3 Using Java to build protocol components

Java 1.1 provides a number of features that facilitate component based software development. These include *inner classes*, Bean *properties*, *serialization* and Bean *events*. These all play an important role in making development of protocols easier.

A basic protocol component, i.e., a conduit, has (at least) two sides. Whenever a message arrives at the protocol component, it is important to know where the message came from, in order to be able to act on the message. On the other hand, it is desirable to view each conduit as a separate unit, having its own identity. Java inner classes and the way the Java Beans architecture uses them, provides a neat solution for this problem.

Each conduit is considered a single Java Bean. Internally the component is constructed from a number of objects: the conduit itself, sides A and B, and typically also some other objects depending on the exact sort of the conduit. The Conduit class itself is a normal Java class, specialized as a Session, Mux, ConduitFactory or such. On the other hand, the side objects, A and B, are implemented as inner classes of the Conduit class. In most respects, these objects are invisible to the rest of the object world. They implement the Conduit interface, delegating most of the methods back to the conduit itself. However, their being separate objects makes the source of a message arriving at a conduit immediately apparent.

Since the conduits are attached to each other, when constructing the conduit graph, the internal side objects are actually passed to the neighbour conduits. Now, when the neighbouring conduit passes a message, it will arrive at the receiving conduit through some side object. This side object uniquely identifies the source of the message, thereby allowing the receiving conduit to act appropriately.

The Java Bean properties play a different role. Using the properties, the individual conduits may publish run time attributes that a protocol designer may use through a visual design tool. For example, the Session conduits allow the designer to set the initial state as well as the set of allowed states using the properties. Similarly, the Accessor object connected to a Mux may be set up using the Beans property mechanism.

Java 1.1 provides a generic event facility that allows Beans and other objects to broadcast and receive event notifications. In addition to the few predefined notification types, the Beans are assumed to define new ones. Given this, it is natural to map conduit messages onto Java events.

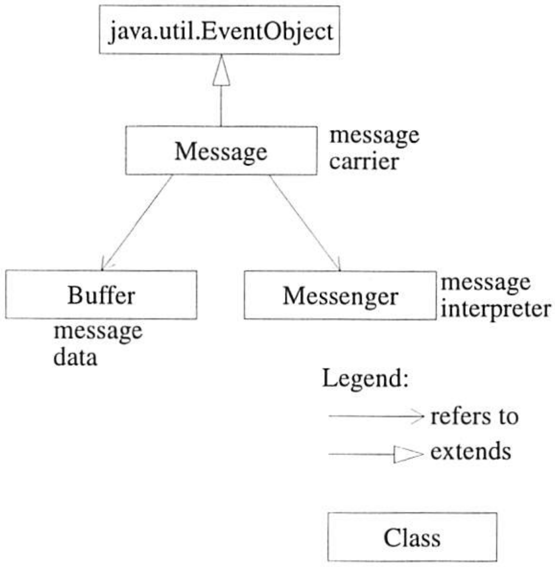


Figure 3: Structure of conduits messages

In Java Conduits, a protocol message is composed of three objects: a *message carrier*, a *message body* and a *message interpreter*. The message carrier extends the `java.util.EventObject` class, thereby declaring itself as a Bean event. The carrier includes references to the message body that holds the actual message data, and a message interpreter that provides protocol specific interpretation of the message data. The message interpreters are called Messengers, and they act in the role of a command according to the Command pattern [10].

Messages are passed from one conduit to the next one using the Java event delivery mechanism. The next conduit registers its internal side object as an event listener that will receive events generated by the previous conduit.

The actual message delivery is synchronous. In practice, the sending conduit indirectly invokes the receiving conduit's accept method, passing the message carrier as a parameter. The receiving conduit, depending on its type and purpose, may apply the Messenger to the current protocol state, yielding an action in the protocol state machine, replace the Messenger with another one, giving new interpretation to the message, or act on the message independent on the Messenger. Typically, the same event object is used to pass the message from conduit to conduit until the message is delayed or consumed.

Java Conduits use the provider / engine mechanism offered by the JDK 1.1 security API. Since neither the encryption / decryption functionality nor its interface specification was not available outside the United States, we created a new engine class `java.security.Cipher` along the model of `java.security.Signature` and `java.security.MessageDigest` classes.

The protocols use the cryptographic algorithms directly through the security API. The data carried in the message body is typically encrypted or decrypted in situ. When the data is encrypted or decrypted, the associated Messenger is typically replaced to yield new interpretation for the data.

3.4 Usage of language level security features

Java offers a number of language level security features that allow a class library or a framework to be secure and open at the same time. The basic facility behind these features is the ability to control access to fields and methods. In Java, classes are organized in packages. A well designed package has a carefully crafted external interface that controls access to both black box and white box classes. Certain behaviour may be enforced by making classes or methods final and by restricting access to the internal features used to implement the behaviour. Furthermore, modern virtual machines divide classes into security domains based on their classloader. There are numerous examples of these approaches in the JDK itself. For example, the `java.net.Socket` class uses a separate implementation object, belonging to a subclass of the `java.net.SocketImpl` class, to provide network services. The internal SocketImpl object is not available to the users or subclasses* of the socket class. The `java.net.SocketImpl` class, on the other hand, implements all functionality as protected methods, thereby allowing it to be used as a white box.

The Java Conduits framework adheres to these conventions. The framework itself is constructed as a single package. The classes that are meant to be used as black boxes are made `final`. White box classes are usually `abstract`. Their behaviour is carefully divided into user extensible features and fixed functionality.

The combination of black box classes, fixed behaviour, and internal, invisible classes allows us to give the protocol implementor just the right amount of freedom. New protocols can be created, but the framework conventions cannot be broken. Nonetheless, liberal usage of explicit interfaces makes it possible to *extend* the framework, but again without the possibility of breaking the conventions used by the classes provided by the framework itself.

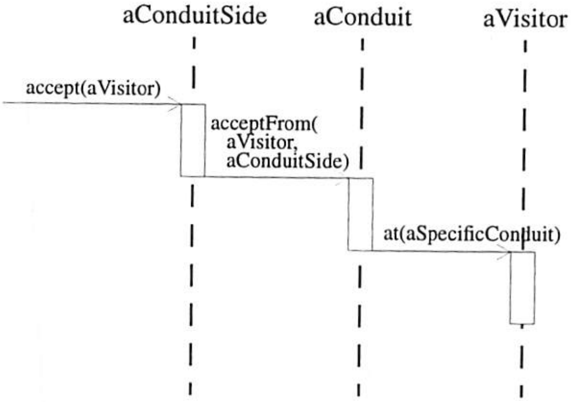


Figure 4: A visitor arrives at a Conduit

All this makes it possible to create *trusted protocols*, and to combine them with untrusted, application specific ones. This is especially important with cryptographic protocols. The cryptographic protocols need access to the user's cryptographic keys. Even though the actual encryption and other cryptographic functions are performed by a separate cryptoengine, the current Java 1.1 security API does not enforce key privacy. However, it is easy to create, e.g., an encryption / decryption microprotocol that encrypts or decrypts a buffer, but does not allow access to the keys themselves.

3.5 Object level design patterns used in the resulting architecture

The Conduits architecture is centred around the idea of a conduit graph that is traversed by protocol messages. The graph is the local representation of a protocol stack. The messages represent the protocol messages exchanged by the peer protocol implementations. This aspect of a graph and graph traversal is abstracted into a Visitor pattern [10]. The pattern is generalized in order to allow also other kinds of visitors to be introduced on demand. These may be needed, e.g., to pass interprotocol control messages or to visualize protocol behaviour.

In this pattern, a protocol message or other visitor arrives as a Java event at an internal side object of a conduit. The side object passes the message to the conduit itself. The conduit invokes the appropriate overloaded at(ConduitType) method of the message carrier, allowing the message decide how to act, according to the Visitor pattern.

As a more complex example of the usage of the gang of four patterns, let us consider the situation when a protocol message arrives at a Session that performs cryptographic functions (see Figure 5). The execution proceeds in steps, utilizing a number of design patterns.

* Actually, other classes within the same package can access the SocketImpl object. Classes outside the package can't.

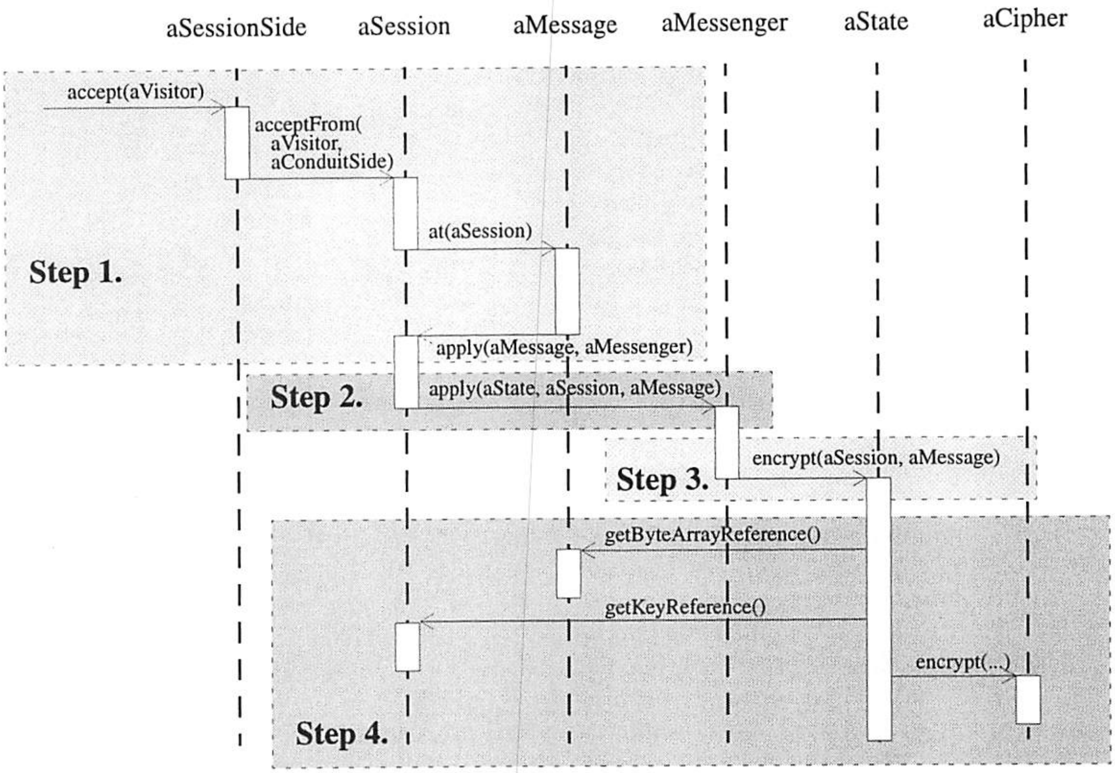


Figure 5: A Message arrives at a cryptographic Session

1. *The message arrives at the Session according to the Visitor pattern.*
   The message is passed to the Session's internal side as a Java Beans visitor event. The event is passed to the session, which invokes the message's at(Session) method. Since the visitor in hand is a message, it calls back the Session's apply(Message) method.
2. *The Session gets the message, and applies it according to the command pattern.*
   The Session uses the Messenger command object, and asks it to be applied on itself, using the current state and message.
3. *The Messenger command object acts on the session, state and message (second half of the Command pattern).*
   This behaviour is internal to the protocol. Typically all states of the protocol implement an interface that contains a number of command methods. The Messenger calls one of these, depending on the message's type. In the example situation where a message arrives and should be sent encrypted, the Messenger invokes the protocol state's encrypt(Session, Message) method.
4. *The current State object acts on the Session and Message.*
   This, again, depends on the protocol. The State may replace the current state at the Session with another State (according to the State pattern), modify the actual data carried by the message, or replace its interpretation by changing the Messenger associated with the Message. In our example, the State encrypts the message data. A reference to a Cipher has been obtained during the State initialization through the Java 1.1 security API. The key objects are stored at the Session conduit.

As examples of other kinds of usage of patterns, the following are worth mentioning:

- The actual encoding/decoding aspect of the Muxes is delegated to separate Accessor objects using the Strategy pattern.
- The State objects are designed to be shared between the Sessions of the same protocol. In order to encourage this behaviour, the base State class implements the basic details needed for the Singleton pattern.
- The ConduitFactories are used as black boxes in the framework. Each ConduitFactory has a reference to a Conduit that acts as its prototype, following the Prototype pattern.
- Obviously, the Adaptor conduits act according to the Adapter pattern with respect to the world outside the conduits framework.
- With respect to the Visitor pattern, the Protocol conduits act according to the Proxy pattern, delegating actual processing to the conduits encapsulated into the protocol.

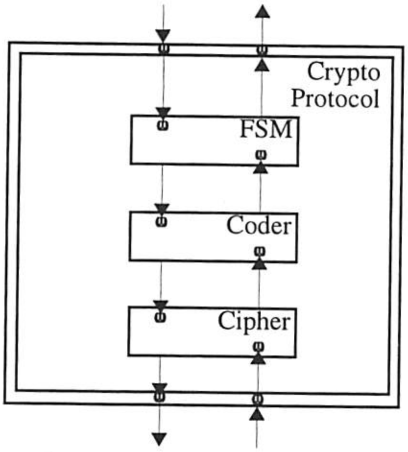


Figure 6: Cryptographic protocol pattern

3.6 Protocol design patterns

Our experience with the framework has shown that protocol independent implementation patterns do arise. That is, there seems to be certain common ways how the different conduits are connected to each other when building protocols. Here we show how the use of encryption tends to be reflected as a conduit topology pattern.

A cryptographic protocol handles pieces of information that are binary encoded and cryptographically protected. Usually the whole message is signed*, encrypted, or both. This yields a highly regular conduits structure where three sessions are stacked on top of each other (see Figure 6). The uppermost session (FSM) receives messages from upper protocols or applications, and maintains the protocol state machine, if any. Directly below lies a session that takes care of the binary encoding and decoding of the message data (Coder). The lowermost session within the protocol takes care of the actual cryptographic functions (Cipher).

According to the conduits architecture, the actual cryptographic keys are stored into the cryptosession. Thus, the information about what key to use is implicitly available from the conduit graph topology. However, this is not always feasible.

In the case of IPSEC AH protocol we resorted to storing the keying information as additional, out of band information within the outgoing protocol message. Similarly, the incoming messages are decorated with information about the security associations that actually were used to decrypt or the check the message integrity. These are then checked further up in the protocol stack to ensure security policy.

4 Implementation experiences

Our current prototype is the third one in a series. The first working prototype was successfully implemented in December 1996. The second one was a complete rewrite, based on the experiences with the first one. The main difference between the second and third prototypes is Java Beans support. The only major change needed was to the message delivery mechanism, due to the added Java event support. Other than that, compliance with the Beans architecture required method naming changes and other minor changes needed to properly support serialization. The protocols themselves were transferred from the second framework prototype to the third with almost no changes. Our next step is to further enhance Java Beans support to facilitate visual protocol composition.

4.1 The framework

The elements used to build protocols are relatively small. This leads to a very piecemeal protocol development. According to our experience, once one is familiar with the model, the actual implementation of protocols is usually very straightforward and fast.

The small component approach seems to be very well suited for building microprotocols. For example, it is easy to represent the individual IPv6 header handlers as separate protocols, and create runtime structures to mix and match them appropriately.

Event delivery and scheduling. The basic Java event delivery mechanism is synchronous. The event source invokes, directly or indirectly, an appropriate method at every registered listener. However, nothing in the architecture mandates this approach. Since events are represented as objects, their delivery may well be queued and delayed. In fact, the listeners themselves may easily create an event queue if desired.

Our current goal is to achieve better performance on a uniprocessor implementation. Earlier experience with a UNIX STREAMS based IPSEC prototype [1] has shown that scheduling should be avoided on a uniprocessor environment. Therefore we have tried to minimize the number of threads and synchronized methods in the current prototype. This may change later when multiprocessing is taken care of.

The framework has one main thread that takes care of carrying a message through the conduit graph. It handles one message a time, passing it from conduit to conduit. If a conduit cannot pass the message, e.g., because it is a partial message and the other fragments are needed, the message stops at the conduit. The carrier thread then

[*] Signed or otherwise integrity protected

handles the next message in queue, or waits if there are no messages currently waiting in a message/event queue.

A separate thread takes care of timers. Timer events are delivered to the conduits by the same thread as the message and other visitor events. A conduit may register a timer event to be scheduled at a particular time, after some delay, or periodically. Whenever the timer expires, a timer event is added to the message/event queue. After the carrier thread has handled a message, it takes the next message or timer event from the queue, and delivers it.

The adapters protect the conduits framework from other threads. The adapter methods are fully synchronized, and may be called by whatever threads. When a message arrives at an adapter from outside, the adapter wraps the message data into a conduit message carrier, attaches some interpretation to it, and places the message into the message/event queue. The carrier thread will select it at first appropriate opportunity.

Since Java I/O is inherently synchronous, the adapters communicating with the world external to the virtual machine typically contain their own internal threads. This allows the conduit processing to continue independent on delays on external I/O.

Memory management. The framework discourages explicit object creation and garbage collection. Typically, the constructors are either private (for black box classes) or protected (for white box classes). Most classes provide a public static instantiation method. This allows objects to be recycled by the class rather than being created and garbage collected for every occasion.

Footprint. The current framework prototype consists of 43 public classes, or about 3800 lines of Java source code (including comments). Only about 760 lines were written by hand; the rest were generated using an UML based case tool.

Of the 43 public classes, 23 are actual user visible classes. The rest are various exceptions (5), housekeeping classes (10) or other classes (5). The relationships of the user visible classes are displayed as an UML class diagram in Appendix A.

4.2 IPSEC

Our IPSEC prototype is designed to work with both IPv4 and IPv6. So far, it has been tested only with IPv6. It is designed to be policy neutral, allowing different kinds of security policies to be enforced.

A basic IP protocol stack, including IPSEC, is shown in Figure 7. In this configuration, the IPSEC is located as a separate protocol above IP. IP functions as usual, forwarding packets and fragments and passing upwards only the packets that are addressed to the current host. IPSEC receives complete packets from IP. The example configuration initially accepts packets that have either no protection, or are protected with AH, or with AH *and* ESP. It does not accept packets that are protected with ESP only or with e.g. double AH. This is one expression of policy. Furthermore, the conduit graph effectively prevents denial of service attacks with multiply encrypted packets.

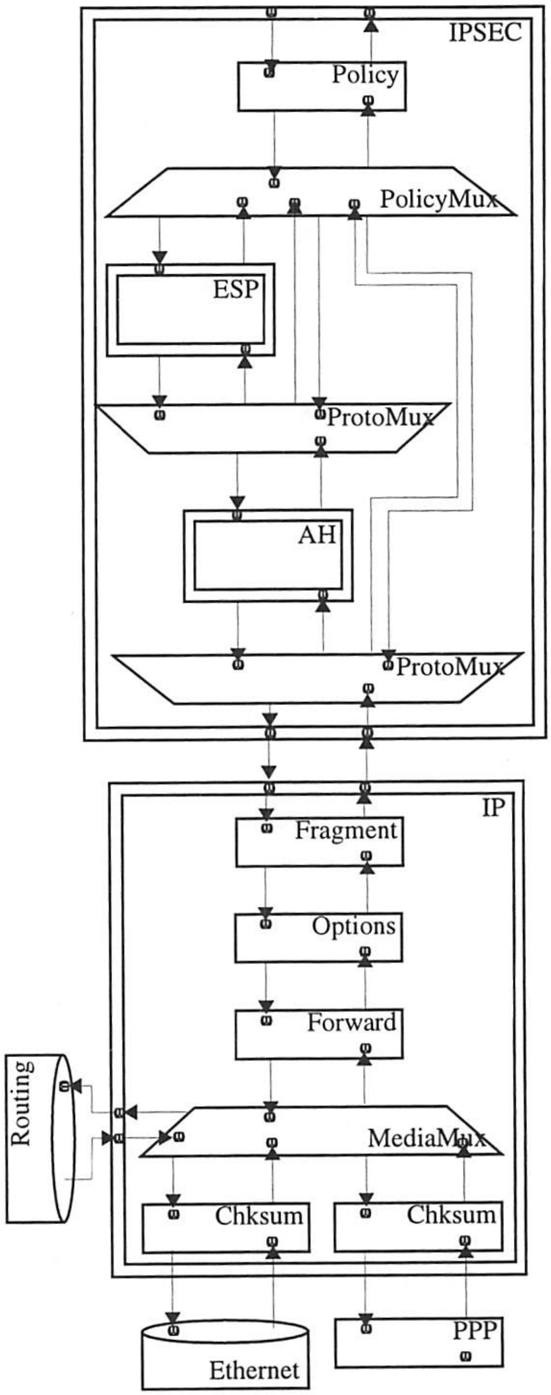


Figure 7: Host IPSEC conduit graph (simplified)

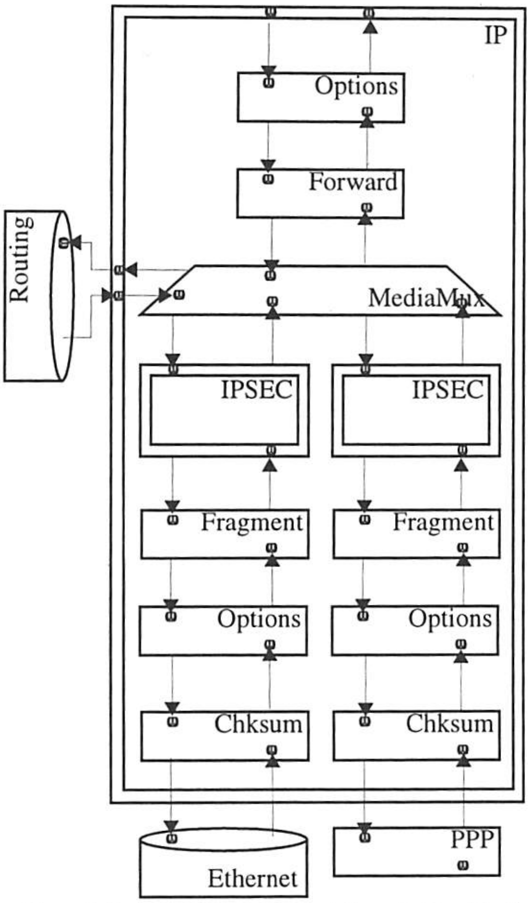


Figure 8: Security GW IPSEC conduit graph (simplified)

During input processing, the AH and ESP protocols decorate the packet with information about performed decryptions and checks. Later, at the policy session, this information is checked to ensure that the packet was protected according to the desired policy. We have also experimented with an alternative configuration, where the policy is checked immediately after every successful decryption or AH check. This seems to be more efficient, since faulty packets are typically dropped earlier. However, the resulting conduits graph is considerably more complex.

During output processing, the policy session and the policy mux together select the right level of protection for the outgoing packet. This information may be derived from the TCP/UDP port information or from tags attached to the message earlier in the protocol stack.

A different IPSEC configuration, suitable for a security gateway, is shown in Figure 8. In this case, instead of being on top of IP, IPSEC is integrated as a module within the IP protocol. Since the desired functionality is that of a security gateway, we want to run all packets through IPSEC and filter them appropriately. Since IPSEC is always applied to complete packets, all incoming packets must be reassembled. This is performed by the Fragment session, which takes care of fragmentation and reassembly.

Once a packet has travelled through IPSEC, passing the policy decisions is applies, it is routed normally. Packets destined to the local host are passed to the upper layers. Forwarded packets are run again through IPSEC, and a separate outgoing policy is applied to them. In this case, it is easier to base the outgoing policy on packet inspection rather than on separate tagging.

Our current IPSEC prototype runs on top of our IPv6 implementation, also built with Java Conduits, on Solaris. We use a separate Ethernet adapter, which is implemented as a native class on top of the Solaris DLPI interface. We have not yet applied JIT compiler technology, and therefore the current performance results are modest.

4.3 ISAKMP

The structure of our ISAKMP implementation is shown in Figure 9. The implementation is attached to the Java UDP implementation through a UDP adapter. Alternatively, it could be attached directly on top of our own UDP implementation. On top of the ISAKMP implementation lies a security policy manager, which forms the "political layer" of our protocol stack.

ISAKMP packets received through the UDP adaptor are directed either to an ISAKMP factory or to some established ISAKMP session, depending on the ISAKMP cookies. If the packet initiates a new ISAKMP association (i.e., is the first main mode packet), the ISAKMP factory consults the upper layer to determine whether the association should be established. The same applies for proposals for new AH or ESP associations. If a new AH or ESP association is accepted by the policy, the AH/ESP factory creates a new AH or ESP protocol instance. The protocol instance takes care of running the ISAKMP quick mode to create the new association.

When a new AH or ESP association has been established, the negotiated parameters are passed to the policy layer. The policy manager takes care of creating the new association to the IP stack, either through PF_KEY interface (if a non-conduits IPSEC is used), or by modifying the IP/IPSEC conduit graph appropriately.

The main novelty in our approach is the separation of the ISAKMP daemon and the policy manager. Currently the policy manager is implemented as a separate conduits protocol. However, it would be possible to implement the policy manager outside the conduits framework as well, and use Java events to communicate between the conduit world and the policy manager.

The current implementation is slightly out of date, due to changes recently made to the ISAKMP and Oakley Internet drafts [19].

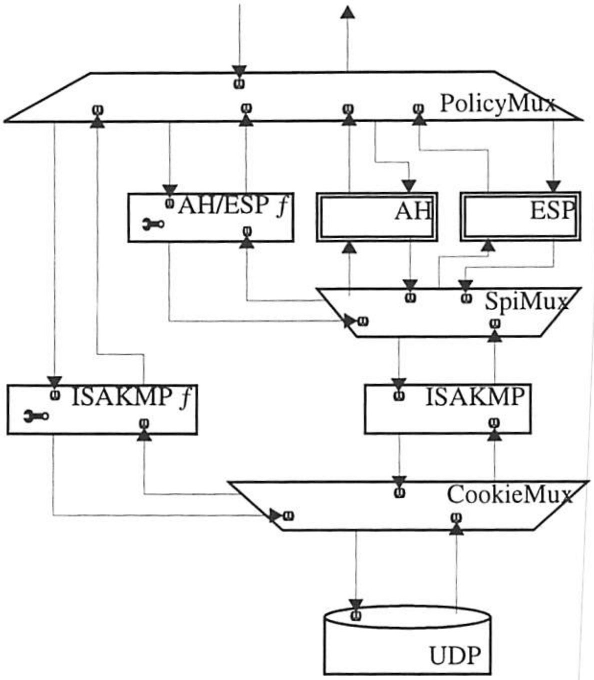


Figure 9: ISAKMP conduit graph (simplified)

4.4 Non-cryptographic protocols

In addition to the cryptographic protocols, we have implemented partial but functional prototypes of the IPv4, IPv6, ARP, ICMP (IPv4 version only), UDP and TCP protocols. Integration of these, along with the IPSEC implementation, into a complete TCP/IP protocol stack is under way.

4.5 Availability

The current framework prototype is available at `http://www.tcm.hut.fi/~pnr/jacob/`. The actual protocol prototypes and the protocol sandbox prototype are available directly from the authors. An integrated, JDK 1.2 based release is expected to be published in late May or early June.

5 Summary

We define an architecture and an object oriented implementation framework for cryptographic protocols. The architecture is based on the Internet, WWW, Java and an initial security context, and optionally augmented with a PKI and the ISAKMP and IPSEC protocols. The implementation framework is based on a fully object oriented language, so it benefits greatly from design patterns, making it easy to use and extensible at the same time. Furthermore, the use of object level design patterns leads to a highly stylistic way of implementing protocols, thereby allowing creation of new, higher level *protocol patterns*.

The implementation framework was developed with JDK 1.1 using the Java Beans and the security API of Java 1.1. In the framework, protocols are built from lower level component called conduits. The protocols are conduits themselves, allowing incremental building of higher level protocols from lower level ones.

The Java execution environment allows the resulting protocols to be seamlessly integrated into the operating system and applications alike. This is especially important for security protocols, since this allows the security systems at various levels to be integrated. We have taken advantage of the Java language level security features (packages, visibility, classloaders). The framework is implemented as a single Java package. Special attention has been paid to dividing the functionality into fixed and user customizable feature sets.

So far we have implemented functional prototypes of IPv4, IPv6, ARP, ICMP, UDP, TCP, IPSEC and ISAKMP protocols. We expect to implement prototypes of further protocols in the near future.

6 Future work

There are a number of future projects that we are planning to start. Due to our limited resources we have not been able to work on all the fronts simultaneously.

Even though a PKI is not an absolute prerequisite for using our architecture, it is in practice essential for most wide-spread real-life applications. We are currently implementing SPKI type certificates that will be integrated into our framework.

The use of security services and features is usually mandated by security policies. The management of security policies in global networks has become a major challenge. We have recently started a project to design and implement an Internet Security Policy Management Architecture (ISPMA) based on trusted Security Policy Managers (SPM). When a user contacts a service, they need to be authorized. Authorization may be based on the identity or credentials of the user. Having obtained the necessary information from the user, the server asks the SPM if the user can be granted the kind of access that they have requested. Naturally all communications between the parties need to be secured.

A graphical Java Beans editor could make the work of the implementor much more efficient than it currently is. This would also make it easier to train new, on the average only average, programmers to develop secure applications. In a graphical editor, the building blocks of our architecture would show as graphical objects that can be freely combined into a multitude of applications. The amount of programming work in developing such an editor is quite large and there certainly are lots of ongoing projects in the area of graphical Java Beans editors. Our plan is to take an existing editor and integrate it into our environment.

So far our work has been focused on the design and implementation of secure application specific protocols. Our long term goal is to create an integrated development environment for entire secure applications. This environment would also include tools for creating the user interface and database parts of the applications.

References

[1] Timo P. Aalto and Pekka Nikander, "A Modular, STREAMS Based IPSEC for Solaris 2.x Systems", In *Proceedings of Nordic Workshop on Secure Computer Systems,* Goethenburg, Sweden, November 1996.

[2] Robert Allen and David Garlan, "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology,* 6(3), July 1997.

[3] Ross J. Anderson, "Programming Satan's Computer", In *Computer Science Today — Recent Trends and Developments*, LNCS 1000, pp. 426–440, Springer-Verlag, 1995.

[4] Ross J. Anderson and Roger Needham, "Robustness principles for public key protocols", *Advances in Cryptology—CRYPTO'95 Proceedings*, Springer-Verlag, 1995.

[5] Ken Arnold and James Gosling, *The Java Programming Language*, Addison-Wesley, 1996.

[6] Randal Atkinson, *Security Architecture for the Internet Protocol,* RFC1825, Internet Engineering Task Force, August 1995.

[7] Kent Beck and Ralph Johnson, "Patterns Generate Architectures", In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'94)*, Bologna, Italy, pp. 139–149, Springer-Verlag, 1994.

[8] Kenneth Birman and Robert Cooper, "The ISIS Project: Real Experience with a Fault Tolerant Programming System", *Operating Systems Review,* pp. 103–107, April 1991.

[9] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas and Tatu Ylönen, *Simple Public Key Certificate*, Internet-Draft `draft-ietf-spki-cert-structure-02.txt`, work in progress, Internet Engineering Task Force, July 1997.

[10] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns — Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[11] Benoit Garbinato, Rachid Guerraoui, "Using the Strategy Design Pattern to Compose Reliable Distributed Protocols", The *Third Conference on Object-Oriented Technologies and Systems (COOTS) Proceedings*, Portland, Oregon, June 16-20, 1997, pp. 221–232.

[12] Li Gong, *Java Security Architecture (JDK1.2) DRAFT DOCUMENT (Version 0.7)*, Sun Microsystems, October 1, 1997, `http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-spec.doc.htm`

[13] Nevin Heintze and J. D. Tygar, "A model for secure protocols and their compositions", In *Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy,* pp. 2–13, IEEE Computer Society Press, May 1994.

[14] Herman Hueni, Ralph Johnson, R. Angel, "A framework for network protocol software", *Object Oriented Programming Systems, Languages and Applications Conference Proceedings (OOPSLA'95)*, ACM Press 1995.

[15] N. C. Hutchinson and L. L. Peterson, "The x–Kernel: An architecture for implementing network protocols." *IEEE Transactions on Software Engineering,* 17(1):64–76, January 1991.

[16] Darrell Kindred, Jaennette M. Wing, "Fast, Automatic Checking of Cryptographic Protocols", In *Proceedings of the Second USENIX Workshop on Electronic Commerce,* November 18-21, 1996, Oakland, California.

[17] Wenbo Mao and Colin A. Boyd, "Development of authentication protocols: some misconceptions and a new approach", *Proceedings of IEEE Computer Security Foundations Workshop VII*, IEEE Computer Society Press, 1994, pp. 178-186.

[18] S. W. O'Malley, L. L. Peterson, "A Dynamic Network Architecture", *ACM Transactions on Computer Systems* 10(2):110–143, May 1992.

[19] Douglas Maughan, Mark Schertler, Mark Schneider and Jeff Turner, *Internet Security Association and Key Management Protocol (ISAKMP),* Internet-Draft `draft-ietf-ipsec-isakmp-08.txt`, work in progress, Internet Engineering Task Force, July 1997.

[20] Bertrand Meyer, "The Next Software Breakthrough", *Computer,* 30(7): 113–114, IEEE Computer Society, July 1997.

[21] Pekka Nikander, *Modelling of Cryptographic Protocols,* Licenciate's Thesis, Helsinki University of Technology, December 1997.

[22] H. Orman, S. O'Malley, R. Schroeppel, and D. Schwartz. "Paving the road to network security, or the value of small cobblestones". In *Proceedings of the 1994 Internet Society Symposium on Network and Distributed System Security*, February 1994.

[23] Michael K. Reiter, Kenneth P. Birman and Robbert Van Renesse, *A Security Architecture for Fault-Tolerant Systems*, Cornell University Technical Report, TR93-1354, June, 1993.

[24] Robbert van Renesse, Kenneth P. Birman and Silvano Maffeis, "Horus, a flexible Group Communication System," *Communications of the ACM*, April 1996.

[25] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr, "A Framework for Protocol Composition in Horus", In *Proceedings of Principles of Distributed Computing*, August, 1995.

[26] Jorma Rinkinen, *Java DES Speed Test,* `http://www.tcm.hut.fi/~jrin/des/` July 1997.

[27] Aviel D. Rubin and Peter Honeyman, *Formal methods for the analysis of authentication protocols*, Technical Report 93–7, Center for Information Technology Integration, Department of Electrical Engineering and Computer Science, University of Michigan, 8. November 1993.

[28] Douglas C. Schmidt, "Using Design Patterns to Develop Reusable Object-Oriented Communication Software", *Communications of the ACM*, 38(10):65–74, October 1995.

[29] Gustavus J. Simmons, "Cryptanalysis and protocol failures", *Communications of th*e ACM, 37(11):56–65, November 1994.

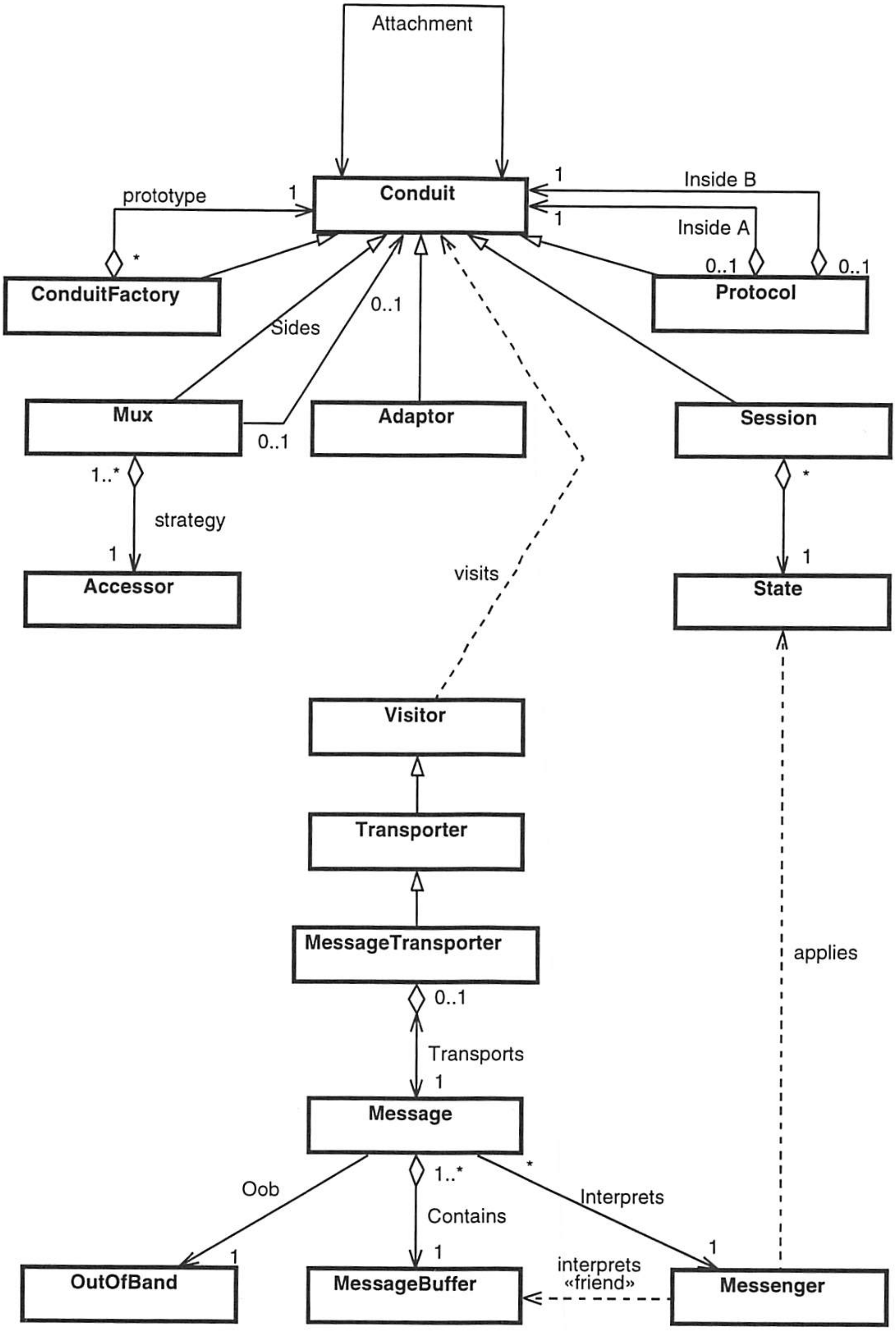
[30] R. Thayer, N. Doraswamy and R. Glenn, IP Security Document Roadmap, Internet-Draft `draft-ietf-ipsec-doc-roadmap-01.txt`, work in progress, Internet Engineering Task Force, July 1997.

[31] Amy Moormann Zremski and Jeannette M. Wing, "Specification Matching of Software Components", *ACM Transactions on Software Engineering and Methodology*, 6(4), October 1997.

[32] Jonathan M. Zweig and Ralph E. Johnson, "The Conduit: A Communication Abstraction in C++", In *Usenix C++ Conference Proceedings,* San Francisco, CA, April 9–11, 1990, pp. 191–204. The Usenix Association 1990.

[33] Joanne Wu (Editor), *Component-Based Software with Java Beans and ActiveX*, White paper, Sun Microsystems, `http://www.sun.com/javastation/whitepapers/javabeans/javabean_ch1.html`, August 1997.

Appendix A UML class diagram



Secure Videoconferencing

Peter Honeyman
Andy Adamson
Kevin Coffman
Janani Janakiraman
Rob Jerdonek
Jim Rees

Center for Information Technology Integration
University of Michigan
Ann Arbor

Abstract

At the Center for Information Technology Integration, we are experimenting with algorithms and protocols for building secure applications. In our security testbed, we have modified VIC, an off-the-shelf videoconferencing application, to support GSS, a generic security interface. We then layered these interfaces onto a smartcard-based key distribution algorithm and a fast cipher. Because these components are accompanied by rigorous mathematical proofs of security and are accessed through narrowly-defined interfaces, we have confidence in the strength of the system's security.

1. Introduction

Security and cryptography research and development are advancing at an accelerating rate, yet the payoff in secure distributed applications is not being realized [1, 2]. This failure is due in part to limitations in Internet infrastructure, such as secure naming and routing, which are not to be found except in isolated prototypes.

Progress is being made in securing the essential fabric of the Internet [3, 4, 5], but even these efforts may fail to meet the security needs of the most stringent distributed applications, which must rely on end-to-end methods to satisfy their exceptional security requirements. Security middleware — protocols, algorithms, and interfaces — supports the design and construction of secure distributed applications and provides a context in which the underpinnings of network security can be explored and customized.

At the Center for Information Technology Integration, we have built a security testbed that supports prototyping and experimenting with secure distributed applications by providing interfaces to secure communications facilities. The testbed is rich in protocols, ciphers, and other middleware for secure computing; provides for rapid prototyping of secure applications by supporting standards-based interfaces; and allows extensive and flexible performance measurement.

In this paper, we describe a characteristic prototyping activity in the CITI testbed: a secure videoconferencing application. This application has interfaces to the user through a standard graphical user interface, and to peer videoconferencing applications through network communications. We assume good host security, a requirement for secure user interaction, and narrow our attention to threats on networks and remote hosts.

Our approach to building a secure videoconferencing application is to layer its communications needs on secure middleware, using standard interfaces to security functionality (such as encrypting a message or establishing a key) and to communications mechanisms (such as establishing a streaming, half-duplex video interface with a peer). We use a freely available Internet videoconferencing application, to which we add a collection of ciphers and key distribution protocols. The interface between the two is defined with the Internet Generic Security Services standard.

We have good confidence in the security of our videoconferencing prototype. We implement critical cryptographic functionality on secure hardware and access the smartcard with a provably secure key distribution protocol. We hand the session keys that result to a provably secure stream cipher. Cryptographic functionality is accessed through GSS, the sole interface to the cryptographic modules.

The remainder of this paper is organized as follows. In Section 2, we describe the videoconferencing application that we have adapted for secure communications. Section 3 describes the ciphers that meet the performance and security requirements of network videoconferencing. Section 4 details the mechanisms we use for secure key distribution and the role of smartcards. In Section 5, we discuss the interface standard that ties the cryptographic methods to the application. Section 6 describes our experiences with the system and the testbed, and Section 7 discusses future plans.

2. Videoconferencing

With its extreme CPU and I/O performance demands, videoconferencing is a distributed communications service that provides plenty of opportunity for performance measurement and tuning. Consider the data requirements of moving color video images over a data network. Working with full-motion (30 frames per second), 24-bit color, 320×240 images, we need to move almost 7 MBps from the camera, through computers and networks, to the screen. This degree of performance is impossible in most of today's Internet, so we must compress the video stream. Then we need to encrypt it.

We have had good experience with hardware motion JPEG encoders running on IBM workstations, even though these encoders are designed for video capture and storage, not videoconferencing. Nonetheless, we are drawn to the commodity PC market to meet our computing needs. We don't find the fastest computers there, we find the cheapest. Because these computers obey Moore's law, they are increasing exponentially in speed at a constant price.

Cheap MPEG decoders for PCs and laptops are common, thanks to a thriving audio/video playback market. Hardware MPEG encoders remain bulky and expensive while software MPEG encoding demands more cycles than our PCs can provide. We anticipate that the next few years will feature fast, cheap laptops equipped with hardware compression and decompression, but the current state of affairs obeys the maxim ''select two from {good, fast, cheap}.'' So with PCs, encoding is either good or fast; with IBM RS6Ks, we get both, but at considerably higher cost.

As a starting point for our secure videoconferencing prototype, we chose VIC [6], the popular and sophisticated MBONE videoconferencing tool. VIC implements user interface management in Tcl/Tk [7], so it is flexible and easy to extend. VIC offers optional DES encryption of the video stream, but leaves key distribution to users. VIC also includes a weak cipher that XORs the video data stream with a constant key value. This is not secure but provides a best-case performance baseline.

3. Ciphers

DES is not the best choice for encrypting video data. The algorithm is strong enough — it has withstood concentrated attack for over 20 years — but the 56-bit DES key space is fast succumbing to exhaustive search by ever-faster processors. DES is also difficult to implement efficiently in software.

We added two ciphers to VIC: RC4 (see Schneier [8]), a simple stream cipher that is reputed to be fast and secure, and VRA [9], a stream cipher invented by Bellcore cryptographers. VRA is not very widely known, so we describe its operation here.

VRA uses a DES-based Goldreich-Levin [10] pseudorandom number generator (PRNG) as an initial source of random bits. Goldreich-Levin is expensive, requiring one or more calls to DES for each outpit bit, so VRA ''stretches'' these bits into a much longer sequence in two ways. The resulting sequence of pseudo-random bits is then XORed into the data stream.

The first stretching technique uses a long, wide table filled with random bits. A subset of the rows of the table is selected at random and combined with XOR. By selecting in advance the total number of rows n and the number of rows selected at random k, the difficulty of recovering the rows from their XOR sum can be made proportional to a desired $\binom{n}{k}$.

The key to effective stretching is to precompute a wide table, so that a lot of bits are produced from a few calls to Goldreich-Levin. In our application, we use a table with 256 rows, 2,048 bits per row. Initializing this table is expensive, but once built a table can be used without limit in multiple sessions.

This stretching technique exhibits good short-term randomness, with a key strength of approximately $\log(n^k)$, or 48 bits for our choices of n and k, but, like any PRNG, admits a birthday attack [11] that effectively halves the key strength.

To compensate for these long-term correlations, VRA uses a second stretching technique, based on random walks through *expander* graphs. Intuitively, this is a family of sparse graphs with ''dense'' interconnectivity. (A sparse graph is one in which the ratio of edges to vertices is upper bounded by a constant.) By dense interconnectivity we mean that for any division of the vertices into equal-sized subsets, the ratio of the number of edges between them to the number of vertices is lower bounded by a constant.

The essential property of expander graphs is that a short random walk in an expander graph arrives at a truly random node. Specifically, if we start at any of the graph's n vertices and take $\log(n)$ random steps, then the final vertex is very nearly equally likely among all the vertices. A huge value for n foils birthday attacks.

Such a graph is enormous but VRA uses Gabber-Galil expander graphs [12], which can be computed on-the-fly as random steps are made. This ability obviates construction of the entire graph, which is utterly infeasible, and allows the procedure to maintain minimal state, just the neighborhood it is currently traversing. The Gabber-Galil graph we use has $2^{1,024}$ nodes, each node having six neighbors.

To avoid making too many Goldreich-Levin calls, each node on the path of the pseudo-random walk is used as output, producing $\log(n)$ pseudo-random bits at each step. This concession to performance certainly exhibits some short-term correlations, but any outputs more than $\log(n)$ steps apart are essentially independent.

The table and graph techniques produce two streams of pseudo-random bits, one with good short-term characteristics, the other with good long-term ones. These bit streams are XORed together, each masking the others weaknesses. The resulting stream is the ultimate output of the VRA PRNG.

VRA is a keyed PRNG. The key is the set of bits used to initialize Goldreich-Levin, and can be of any size. VRA has essential cryptographic properties, is based on concrete mathematical arguments, and passes numerous tests of randomness, including Knuth's multidimensional tests and Marsaglia's Diehard battery of tests (see [9].) Furthermore, and of utmost importance for our videoconferencing application, VRA is fast.

4. Session keys

Communicating peers establish a security context by agreeing on a shared secret, or *key*, that they use to authenticate and secure subsequent communications. If all principals in a security domain must exchange keys in advance, then the number of keys that must be set up grows quadratically with the number of principals. This does not scale well. The additional requirement that all principals manage a private database of keys makes even small scale deployment uncomfortable.

Needham and Schroeder address these complexities by establishing one long-term key for each of the principals in the security domain and sharing the long-term keys with a trusted third party [13]. This approach has two distinct advantages. First, the number of long-term keys in the system grows linearly with the number of principals, not quadratically. Second, each principal is responsible for only the one key that it shares with the trusted third party, not the keys for all of the other principals in the security domain.

While reliance on a trusted third party reduces the obligations and bookkeeping for principals, it does not eliminate their responsibilities altogether, nor shield them from harm in the event that control over a long-term key is lost. To assist principals in the secure management of their keys, researchers at Bellcore devised an innovative key distribution protocol that exploits the tamper-resistant properties of smartcards to provide a convenient and secure repository for cryptographic keys.

4.1. Smartcards

In the systems we use daily, we find the greatest security threat to be the reliance on passwords selected by users. Users pick passwords that are easy to guess [14, 15]. This is especially troublesome in an environment like ours, which relies heavily on Kerberos IV [16] for basic security services; regrettably, Kerberos IV admits an offline dictionary attack [17]. Smartcards are an attractive technology for reducing or eliminating the reliance on weak, user-selected passwords.

A *smartcard* [18] is a plastic card the size and thickness of an ordinary credit card with electrical contacts and an embedded microprocessor. Putting a computer in everyone's hip pocket creates an infrastructure that enables a huge range of applications, such as vending, personal telecommunications, medical information, home banking and ATM, satellite TV, FAX scrambling, *etc.* The development of smartcard infrastructure provides a context for forward-looking projects such as Xerox PARC's ''Ubiquitous Computing'' initiative. [19].

Smartcards are prevalent in Europe and some other parts of the world, but are an emerging technology in North America. European manufacturers such as SGS-Thomson, Siemens, Gemplus, and Schlumberger are among the most prominent, but Motorola and Texas Instruments are also major players.

The introduction of phonecards over a decade ago paved the way for broad acceptance of smartcards in Europe. European banking and merchant industries have also embraced smartcards, using them in applications such as vending, loyalty card, electronic purses, pay-TV, and identification. By the end of 1997, over a billion smartcards, mostly simple phonecards, were in use worldwide, over 50 million of them advanced, microprocessor-equipped cards [20].

Standardization of smartcard physical characteristics and access protocols [18] plays a vital role in applicability and acceptance. The Europay-MasterCard-Visa and Mondex specifications for smartcard payment systems make it likely that smartcards will continue to play an increasing role in European private and public sectors. The engagement of non-European partners paves the way for global acceptance of smartcards.

A typical smartcard contains an 8-bit microprocessor clocked at 5 Mhz with 8K of EEPROM and a few hundred bytes of RAM, communicating at 9.6 Kbps. Fast DES encryption has long been available [21], and arithmetic co-processors are beginning to be used to provide for subsecond public key operations [22, 23].

Most smartcards have advanced security features to protect the contents of memory from being read or altered by unauthorized users and to protect against improper execution of embedded software. These controls typically include design circuitry to ensure that the embedded software either executes correctly or stops in a safe condition. Critical parameters such as supply voltage, clock frequency, and other critical signals are continuously monitored and filtered to avoid faulty execution [24].

Physical constraints limit the applicability of smartcards in settings that require high-speed computing or communication. But unique security and mobility characteristics make them an attractive foundation for deploying secure distributed applications. For example, smartcards may play the vital role of a trusted computing base in applications that employ downloaded executable content [25], such as Java applets.

4.2. Shoup-Rubin protocol

Before videoconferencers can use VRA, or any other cipher for communications privacy, they need to agree on a session key. Bellcore's Shoup-Rubin protocol [26] is a smartcard-based key distribution protocol that runs among two communicating videoconferencers and a trusted third party. Following Schneier, we call these ALICE, BOB, and TRENT [8].

Shoup-Rubin stores long-term keys on smartcards and performs all cryptography necessary for session key distribution on the smartcards. ALICE never knows her long-term key; it is known only to TRENT and to ALICE's smartcard, where it is used as a key in cryptographic computations.

The role of the Shoup-Rubin protocol is to provide fast and secure session key distribution. The session keys distributed with Shoup-Rubin are not stored on secure hardware, and may be vulnerable to compromise; good practice dictates frequent rekeying.

The details of the Shoup-Rubin protocol are fairly intricate, in part to satisfy the requirements of an underlying complexity-theoretic proof framework. This inconvenience is balanced by the ability to prove powerful properties of the protocol. This mathematical strength, coupled with the hardware basis of long-term key storage, lends good confidence in the overall security of the session key distribution mechanism.

Shoup-Rubin builds on the Leighton-Micali key distribution protocol [27], an inexpensive symmetric key distribution protocol. Leighton-Micali uses a construct known as a *pair key* to establish a shared secret between communicating parties.

Let **A** and **B** denote unambiguous identifiers for ALICE and BOB, and let K_A and K_B be their long term keys, and let $\{M\}_K$ denote message M encrypted with key K. ALICE and BOB's *pair key* is defined

$$\Pi_{AB} = \{\mathbf{A}\}_{K_B} \oplus \{\mathbf{B}\}_{K_A}$$

TRENT calculates pair keys on demand; that is TRENT's entire role. Because a pair key reveals nothing about the long-term keys used in its calculation, it can be communicated in the clear.

With pair key Π_{AB} in hand, ALICE computes $\{\mathbf{B}\}_{K_A}$. Combining this with the pair key yields $\kappa = \{\mathbf{A}\}_{K_B}$. BOB can compute κ directly, so once ALICE has a pair key in hand, she and BOB can communicate privately using key κ.

In Shoup-Rubin, κ is computed on ALICE's and BOB's smartcards. ALICE and BOB then use κ to secure the messages that provide for session key agreement.

The Shoup-Rubin protocol is detailed in the Appendix. Shoup and Rubin use Bellare and Rogaway's innovative complexity theoretic techniques [28] to prove that their key distribution algorithm does not disclose the session key, even to an extremely powerful adversary.

4.3. Shoup-Rubin implementation

The Shoup-Rubin protocol is a distributed computation involving five processing elements: ALICE's computer, her smartcard, BOB's computer, his smartcard, and TRENT. TRENT has access to long-term keys for all the principals in the system.

Working with Personal Cipher Card Corp., a smartcard vendor in Lakeland, FL, CITI implemented the smartcard functionality of Shoup-Rubin on the SGS-Thomson ST16612 card, which contains a MC68HC05 microprocessor clocked at 3.58 Mhz containing 2 KB EEPROM, 6 KB ROM, and 160 bytes RAM. The card supports DES encryption, so that is what we use. Each smartcard call takes about 300 msec. The Shoup-Rubin implementation is about 500 bytes of code, stored in EEPROM.

The total time for key distribution, from the moment a smartcard is inserted into a reader to the time when keys are available, is about 10 seconds. This lengthy delay is due in part to deficiencies in our Windows95 smartcard drivers, but also reflects the message overhead of navigating the ISO 7816 file system on the card. Our goal is one or two seconds on average.

5. Interfaces

To make the encryption and key exchange algorithms available for use in VIC and other applications, we built a Generic Security Service (GSS) [29] interface encompassing the four ciphers (DES, XOR, RC4, and VRA) and Shoup-Rubin. As the name implies, GSS provides security services to callers in a generic fashion, allowing applications to be written to a common portable interface. GSS may be implemented with a range of underlying mechanisms.

The GSS API has four categories of interfaces: credential management, security context, message operations, and support. Shoup-Rubin keeps its credentials on smartcards, so our interface does not implement credential management. A handful of support calls were implemented to handle buffer management and naming issues. Security context establishment and message operations constitute the bulk of our GSS implementation.

We implemented `GSS_Init_sec_context` and `GSS_Accept_sec_context`. The security context interface provides key exchange and establishment of a security context, *i.e.*, the session key, between two entities. The calling applications use the GSS API without knowledge of the underlying mechanisms being used. They call `GSS_Init_sec_context` or `GSS_Accept_sec_context` and pass opaque tokens back and forth until the status values returned indicate that the processing is complete.

We implemented the `GSS_Wrap` and `GSS_Unwrap` message calls. These routines provide for data confidentiality by encrypting the input data. We use the quality of protection parameter, or QoP, to select among encryption methods.

We extended VIC to make GSS calls and augmented its Tcl/Tk interface to allow online cipher selection and performanced data capture. This lets us demonstrate and measure how the choice of ciphers affects the quality of the delivered video.

Implementation of TRENT presents some challenges. On the one hand, TRENT must be online and available at all times. On the other hand, TRENT is entrusted with all of the long-term keys in the system. Combined, these requirements make TRENT a high profile and vulnerable target. We use smartcards to provide for TRENT's seemingly contradictory requirements.

TRENT's function is directory service, so we use an off-the-shelf Lightweight Directory Access Protocol (LDAP) [30] server to provide a standard interface for pair key requests and responses. To minimize the security requirements of the LDAP server, we encrypt the long-term keys with a master key before storing them on the server, and use an attached smartcard for the actual pair key computation. With this approach, the pair-key service can be hosted on a server with security requirements comparable to an email or web server, rather than the extremely stringent security requirements that would be anticipated for a network-attached server holding such vulnerable assets as long-term keys.

6. Testbed Assessment

The CITI security testbed, consisting of a collection of ciphers and key distribution methods tied together with Internet-standard interfaces, supports the development of secure applications. The testbed is easy to extend, and we anticipate adding building blocks.

Our extensions to VIC yield a videoconferencing tool with standard security interfaces, provably secure key distribution, and provably secure end-to-end encryption. Our ability to build, demonstrate, and instrument a prototype implementation validates the usefulness of our security testbed. The tool itself remains very portable and efficient.

Performance measurements were taken from 166 Mhz Pentium systems running Windows '95. The bottlenecks are video encoding and decoding, and Wintel data movement. In these experiments, the presence or absence of encryption makes no difference in throughput. Nonetheless, we are able to measure the time spent in the encryption functions, and thus can estimate the throughput we might expect once we solve our video bottleneck.

Cipher	Throughput
XOR	17 MBps
VRA	3.8 MBps
RC4	1.5 MBps
DES	0.6 MBps

We are encouraged to see VRA outpacing RC4, and are continuing to tune VRA for Wintel.

Our experiences with IBM hardware have been more encouraging. Because we use hardware codecs, we are able to transmit encrypted video at high speed. The following measurements were taken on IBM RS/6000 Model 42T systems running AIX 4.1.2. These are microchannel-based 122 Mhz PowerPC systems with motion JPEG codecs.

Cipher	Throughput	Frames per sec.
XOR	8.9 MBps	30
VRA	1.48 MBps	30
RC4	0.78 MBps	21
DES	0.57 MBps	17

Cipher performance on RS6K is about half that of Pentium because of byte-ordering overhead for the former architecture.

We have used the RS6K-based videoconferencing system to view and control a scanning electron microscope (SEM) in Ann Arbor from a remote location in Washington, DC. Because we had a dedicated 10 Mbps channel, network delay was negligible and jitter nonexistent. Delays otherwise induced by the system were also negligible, and controlling the SEM was a very satisfactory experience. Using VRA encryption, we were able to sustain full-motion video with negligible performance overhead. Using RC4 or DES, however, caused significant degradation of the frame rate.

To highlight the importance of video stream encryption, we built a network snooper that decodes and displays unencrypted video packets. When encryption is enabled, the packet payloads are no longer recognizable as network video, and the image freezes. While unsophisticated, this simple demonstration of the need for secure communications evokes a positive response in its audience.

7. Future work

In our current and future work, we are extending the security boundaries of VIC to include encrypted audio communications. We are also addressing multiparty communications and the attendant problems in secure and reliable group communications. Reliable multicast offers the potential for efficient key distribution to members of a secure session and can play a central role in secure multiparty communications.

Shoup-Rubin needs a mechanism for revocation of long-term keys. If ALICE's long-term key is compromised, BOB may be tricked into establishing a session with an intruder masquerading as ALICE. This comes about because BOB never communicates with TRENT. We are augmenting Shoup-Rubin to preserve security even when smartcards are compromised.

This is an exciting time to be working with secure tokens: new companies and products are making custom programming of secure tokens easy and fast. We are testing Schlumberger's JavaCard and are implementing and augmenting Shoup-Rubin on that platform. Advances such as this pave the way for us to be able to apply the hardware security inherent in secure tokens in a rapid and direct way.

Acknowledgements

S. Rajagopalan, Bill Aiello, and Scott Dexter provided many illuminations. Piet Maclaine Pont of IBM Netherlands suggested the solution to our TRENT dilemma.

Personal Cipher Card Corp. manufactured our Shoup-Rubin smartcards. We thank Kip Wheeler and PC^3 for working so closely with us.

This work was partially supported by a grant from Bellcore.

Availability

Our modifications to VIC are freely available; contact `info@citi.umich.edu`.

References

1. Matt Blaze, ''If cryptography is so great, why isn't it being used more?,'' Invited talk, USENIX Conference, Anaheim (January 8, 1997).
2. Bruce Schneier, *Why cryptography is harder than it looks*, `http://www.counterpane.com/whycrypto.html`, December 23, 1996.
3. John Gilmore, *Secure Wide Area Network Project*, `http://www.cygnus.com/~gnu/swan.html`, March 20, 1997.
4. D. Eastlake and C. Kaufman, ''Domain Name System Security Extensions,'' RFC 2065, USC/ Information Sciences Institute (January 3, 1997).
5. R. Atkinson, ''Security Architecture for the Internet Protocol,'' RFC 1825, USC/ Information Sciences Institute (August 09, 1995).
6. S. McCanne and V. Jacobson, ''VIC: a flexible framework for packet video,'' in *ACM 3rd Ann. Conf. on Multimedia*, San Francisco (November, 1995).
7. J.K. Ousterhout, ''An X11 toolkit based on the Tcl language,'' in *Proc. 1991 Winter USENIX Conf.*, Nashville (January, 1991).
8. B. Schneier, *Applied Cryptography, Second Ed.*, John Wiley & Sons, New York (1996).
9. W. Aiello, S. Rajagopalan, and R. Venkatesan, ''Practical and provable pseudorandom generators,'' pp. 1–8 in *5th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA)*.
10. O. Goldreich and L.A. Levin, ''Hard core predicates for any one-way function,'' pp. 25–32 in *21st Ann. ACM Symp. on Theory of Computing* (1989).

11. D.R. Stinson, *Cryptography: theory and practice,* CRC Press, Inc. (1995).

12. Ofer Gabber and Zvi Galil, ''Explicit Constructions of Linear-Sized Superconcentrators,'' *JCSS* **22**(3), pp. 407–420 (1981).

13. R.M. Needham and M.D. Schroeder, ''Using Encryption for Authentication in Large Networks of Computers,'' *Communications of the ACM* **21**(12) (December, 1978).

14. Robert T. Morris and Ken Thompson, ''Password Security: A Case History,'' pp. 594–597 in *CACM* (November, 1979).

15. D.V. Klein, ''Foiling the Cracker: A Survey of, and Improvements to, Password Security,'' pp. 5–14 in *Proc. UNIX Security Workshop II*, USENIX Assoc., Portland (August, 1990).

16. J.G. Steiner, C. Neuman, and J.I. Schiller, ''Kerberos: An Authentication Service for Open Network Systems,'' pp. 191–202 in *Winter 1988 USENIX Conference Proceedings*, Dallas (February, 1988).

17. Steven M. Bellovin and Michael Merritt, ''Limitations of the Kerberos Authentication System,'' pp. 253–267 in *Proc. of Winter USENIX Conf.*, Dallas (January, 1991).

18. International Organization for Standardization, ''Identification Cards — Integrated Circuit(s) Cards with Contacts,'' ISO 7816.

19. Mark Weiser, ''The Computer for the Twenty-First Century,'' *Scientific American*, pp. 94–110 (September, 1991).

20. Diogo Teixeira, Cynthia Weaver, and James Beams, ''Smart Cards in Banking: The Future of Money?,'' 1997 Financial Services Technology Conference, The Tower Group. `http://www.towergroup.com/97conf/97conf.htm`

21. Louis Claude Guillou, Michel Ugon, and Jean-Jacques Quisquater, ''The Smart Card: A Standardized Security Device Dedicated to Public Cryptology,'' pp. 561–613 in *Contemporary Cryptology: The Science of Information Integrity*, ed. Gustavus J. Simmons, IEEE Press (1992).

22. David Naccache and David M'Raihi, ''Arithmetic Co-processors for Public Key Cryptography: The State of the Art,'' pp. 39–58 in *Proc. of CARDIS Smart Card Research and Advanced Applications Conf.*, ed. Pieter H. Hartel, Pierre Pardinas, and Jean-Jacques Quisquater, Stichting Mathematisch Centrum (CWI), Amsterdam (Sept. 1996).

23. Ronald Ferreira, Ralf Malzahn, Peter Marissen, Jean-Jacques Quisquater, and Thomas Wille, ''FAME: A 3rd Generation Coprocessor for Optimising Public Key Cryptosystems in Smart Card Applications,'' pp. 59–72 in *Proc. of CARDIS Smart Card Research and Advanced Applications Conf.*, ed. Pieter H. Hartel, Pierre Pardinas, and Jean-Jacques Quisquater, Stichting Mathematisch Centrum (CWI), Amsterdam (Sept. 1996).

24. Antony Watts, ''Smartcards and Security — or How to Save $5 Billion a Year!,'' pp. 102–109 in *Smart Card Technology International*, Chantry Hurst Books, London (1996).

25. TRENT Jaeger, *Flexible Control of Downloaded Executable Content,* PhD Thesis, University of Michigan (1996).

26. V. Shoup and A.D. Rubin, ''Session Key Distribution Using Smart Cards,'' in *Proc. of Eurocrypt '96* (May, 1996).

27. T. Leighton and S. Micali, ''Secret-Key Agreement Without Public-Key Cryptography,'' pp. 456–479 in *Proc. of Crypto '93*, Santa Barbara (1993).

28. M. Bellare and P. Rogaway, ''Provably Secure Session Key Distribution: The Three Party Case,'' in *Proc. ACM 27th Ann. Symp. on the Theory of Computing* (1995).

29. J. Linn, ''Generic Security Service Application Program Interface, Version 2,'' RFC 2078, USC/ Information Sciences Institute (January, 10, 1997).

30. W. Yeong, T. Howes, and S. Kille, ''Lightweight Directory Access Protocol,'' RFC 1777, USC/ Information Sciences Institute (March 1995).

Appendix: Shoup-Rubin details

In this section we give a detailed description of the Shoup-Rubin session key distribution protocol. Initially, ALICE and BOB have smartcards initialized with a secret card key and a long-term key shared with TRENT.

The following table defines the terms used in the Shoup-Rubin smartcard-based session key distribution protocol. Encryption of message M with key K is denoted $\{M\}_K$. Integer operands are concatenated to other protocol terms with the ''dot'' operator to satisfy requirements of the Bellare-Rogaway proof framework.

Term	Meaning
A, **B**	Unique identifiers
K_A, K_B	Long-term keys
K_{AC}, K_{BC}	Secret card keys
r, s	Nonces
$\Pi_{AB} = \{\mathbf{A}\cdot 0\}_{K_B} \oplus \{\mathbf{B}\cdot 1\}_{K_A}$	Pair key
$\alpha = \{\Pi_{AB}\cdot\mathbf{B}\cdot 2\}_{K_A}$	Verifies Π_{AB}
$\beta = \{r\cdot s\cdot 1\}_{\kappa}$	Verifies r and s
$\gamma = \{r\cdot 1\cdot 1\}_{K_{AC}}$	Verifies r
$\delta = \{s\cdot 0\cdot 1\}_{\kappa}$	Verifies s
$\kappa = \{\mathbf{A}\cdot 0\}_{K_B}$	See discussion
$\sigma = \{s\cdot 0\cdot 0\}_{\kappa}$	Session key

The influence of the Leighton-Micali key distribution protocol is evident in the use of ALICE and BOB's *pair key*, defined as

$$\Pi_{AB} = \{\mathbf{A}\}_{K_B} \oplus \{\mathbf{B}\}_{K_A}$$

The pair key allows ALICE and BOB to share a secret without prior agreement.

We now detail the steps of Shoup-Rubin.

From	To	Message	Meaning
ALICE	TRENT	**A**, **B**	ALICE wishes to initiate a session with BOB.
TRENT	ALICE	Π_{AB}, α	Π_{AB} is ALICE and BOB's pair key. α is a verifier for Π_{AB}.

ALICE asks TRENT for the ALICE/BOB pair key. TRENT also returns a verifier, which ALICE's card uses to prevent masquerading.

From	To	Message	Meaning
ALICE	CardA	—	ALICE requests a nonce to verify subsequent communication with BOB.
CardA	ALICE	r, γ	r is a nonce, γ is a verifier for r.

Card operation 1

ALICE initiates the protocol with BOB by requesting a nonce from her smartcard. ALICE retains the verifier for later use.

From	To	Message	Meaning
ALICE	BOB	**A**, r	BOB will use r to assure ALICE of his correct behavior.

By sending a nonce to BOB, ALICE requests establishment of a fresh session key.

From	To	Message	Meaning
BOB	CardB	**A**, r	BOB instructs his smartcard to construct a session key, and provides ALICE's nonce for her subsequent verification.
CardB	BOB	s, σ, β, δ	s is a nonce used to construct the session key. σ is the session key. β is ALICE's verifier for r and s. δ is BOB's verifier for s.

Card operation 2

BOB sends ALICE's identity and her nonce to his smartcard. BOB's card generates a nonce and, from this, a session key. BOB's card also generates two verifiers; one is used by ALICE's card to verify both nonces, the other is used by BOB to verify ALICE's subsequent acknowledgement. BOB retains the session key and his verifier.

From	To	Message	Meaning
BOB	ALICE	s, β	ALICE needs s to construct the session key, and β to verify r and s. BOB retains σ, the session key, and δ, a verifier for s.

BOB forwards his nonce, from which ALICE's card constructs the session key.

From	To	Message	Meaning
ALICE	CardA	**B**, r, s, Π_{AB}, α, β, γ	Verify: Π_{AB} with α, r with γ, and BOB's use of r and s with β. Use Π_{AB} and s to construct the session key.
CardA	ALICE	σ, δ	σ is the session key. δ is sent to BOB to confirm ALICE's verification of s.

Card operation 3

ALICE sends everything she has to her smartcard: BOB's identity, the pair key and its verifier, her nonce and its verifier, and BOB's nonce and its verifier. ALICE's card validates all the verifiers. If everything checks out, ALICE's smartcard constructs the session key from BOB's nonce and uploads it to ALICE along with a verifier to assure BOB that ALICE is behaving properly.

From	To	Message	Meaning
ALICE	BOB	δ	Confirm

ALICE sends the verifier to BOB. BOB compares it to his retained verifier.

Unified Support for Heterogeneous Security Policies in Distributed Systems

Naftaly H. Minsky[*] Victoria Ungureanu[*]

Department of Computer Science
Rutgers University
New Brunswick, NJ 08903
{minsky,ungurean}@cs.rutgers.edu

Abstract

Modern distributed systems tend to be conglomerates of heterogeneous subsystems, which have been designed separately, by different people, with little, if any, knowledge of each other — and which may be governed by *different security policies.* A single software agent operating within such a system may find itself interacting with, or even belonging to, several subsystems, and thus be subject to several disparate policies. If every such policy is expressed by means of a different formalism and enforced with a different mechanism, the situation can get easily out of hand.

To deal with this problem we propose in this paper a security mechanism that can support efficiently, and in a *unified* manner, a wide range of security models and policies, including: conventional *discretionary* models that use capabilities or access-control lists, *mandatory* lattice-based access control models, and the more sophisticated models and policies required for commercial applications. Moreover, under the proposed mechanism, a single agent may be involved in several different modes of interactions that are subject to disparate security policies.

1 Introduction

Modern distributed systems tend to be conglomerates of heterogeneous subsystems, which have been designed separately, by different people, with little, if any, knowledge of each other — and which may be *governed by different security policies.* A single software agent operating within such a system may find itself interacting with, or even belonging to, several subsystems, and thus be subject to several disparate policies. For example, an agent may be subject to a multi-level security policy when retrieving military documents; it may carry *capabilities* that provide it with certain access rights to computing resources; and, while accessing certain financial information, it may be subject to the "Chinese Wall" security policy [4]. If every such policy is expressed by means of a different formalism and enforced with a different mechanism, the situation can get easily out of hand.

To deal with this problem we propose in this paper a security scheme under which policies are defined formally and explicitly, and are enforced by a unified mechanism. Each policy under this scheme specifies the type of messages regulated by it and the *law* that governs these messages.

The proposed mechanism is based on the concept of "law-governed architecture" for distributed systems [13], and on the more recent concept of "regulated interaction" (RI) [14]; it is currently implemented by an experimental toolkit called Moses. This toolkit can support a wide range of security models and policies, including: conventional *discretionary* models that use capabilities or access-control lists, *mandatory* lattice-based access control models [18], and the more sophisticated models and policies required for commercial [5] and clinical [1] applications. Moreover, under Moses, a single agent may be involved in several different modes of interactions that are subject to disparate security policies.

[*] Work partially supported by NSF grants No. CCR-96-26577

The paper is organized as follows: Section 2 attempts to motivate the need for a unified mechanism by considering two security policies that are difficult to support by conventional mechanisms, and which may need to coexist in a single system. Section 3 defines the concept of a security policy under RI, and shows how such policies are defined and enforced. Section 4 presents the detailed implementation under RI of the policies of Sections 2. Sections 5 discusses some related work, and Section 6 concludes this paper.

2 Motivating Examples

To motivate the need for a new unified security mechanism we describe here two policies designed for commercial applications which are difficult to implement by traditional means, particularly in distributed systems; another such policy, involving capabilities, is discussed in Section 3.4. Later we will present the implementation of these policies under the security mechanism to be proposed here, making it evident that a single agent can be subject, concurrently, to several policies.

2.1 The Chinese Wall ($\mathcal{CW}$) Policy

Consider a distributed database that contains files belonging to various commercial companies. Let these companies be partitioned into a set of disjoint "competition cliques," where each clique contains companies that compete with each other in the market place. And let the clients of this database be financial analysts, who may have access to any number of competition cliques. According to a common commercial practice, such access is subject to the *Chinese Wall* ($\mathcal{CW}$) policy [4] which can be stated as follows:

> *A priori, an analyst can get information about* any *company of a clique* q *to which he has access. But once the analyst gets information about a given company in* q*, he is not allowed to get information about any other company in this clique.*

Thus, under this policy, what the analyst can get from the database, at a given moment in time, depends on what he got from the database in the past. Recently, it has been shown how this policy can be implemented in MLS systems by casting it as a multilevel lattice based relabel policy [8], or by using reflexive-flow relations [7]. However, MLS does not lend itself to distributed implementation, where the files of companies in a given clique are maintained by several servers belonging to possible different administrative domains. Moreover, it is very difficult in this case (if at all possible) to implement this policy either by means of *access control lists* (ACL), or by the *capability-based* scheme — the two main access control techniques used in distributed systems [20]. The implementation of the $\mathcal{CW}$ policy using ACL would require each server to know what, if anything, his client got from other servers in the past, or even what he is requesting from them concurrently. This cannot be done in a scalable way, since it requires multi-casting of all queries. The implementation of $\mathcal{CW}$-policy with capabilities would mean that whenever a given client reads one of the files of a company, its capabilities for files of other companies in the same clique should be revoked. But revocation is difficult to carry out by traditional means, in particular because it requires a central authority, and is generally not supported.

2.2 The Sealed-Bid ($\mathcal{SB}$) Auction Policy

A common way for selling artwork or real estate is sealed-bid auction in which secret bids are issued for an advertised item in a predefined time-frame. The security requirements of this process have been studied recently by Franklin and Reiter in [9], and paraphrased here as follows:

1. every auction has a predefined time frame for bidding, no bids can be issued outside of this frame;
2. once a bid is issued it cannot be repudiated; but one can out-bid himself any number of times.
3. the winner is the issuer of the highest bid;
4. the bidders identity are not revealed at any time and to any party, not even to the auctioneer; the same holds true for the sums bided by losing participants.
5. the auctioneer is guaranteed to be able to collect the money from the winning bidder; losing bidders do not forfeit money;

6. an agent can participate concurrently in any number of auctions.

Henceforth we will refer to tnese collection of requirements for sealed-bid auction as the $\mathcal{SB}$-policy.

Observing that this policy does not lend itself to implementation by any of the traditional security mechanisms, Franklin and Reiter proposed an elegant new technique for it. Their implementation uses a novel cryptographic technique called verifiable signature sharing, which requires replicating the auction servers.

We will show in section 4.2 how this policy can be implemented by the very same mechanism that we use to implement the Chinese Wall policy. Moreover, our implementation does not require the duplication of the auction servers, thus being more efficient.

3 Security Policies Under Regulated Interaction

We start by defining our concept of a security policy. We continue by showing how a policy is defined by what is called a "law", and how it is enforced under Regulated Interaction (RI)—describing as much of RI, and of the Moses[1] toolkit that implements it, as is needed for this purpose. In Section 3.4 we illustrate this mechanism by showing how uncopyable capabilities can be implemented under RI. In Section 3.5 we show how policies are created and maintained; and we conclude with a brief discussion of the fault tolerance and scalability of our mechanism.

3.1 The Concept of a Security Policy

We define a *security policy* $\mathcal{P}$ to be a triple $\langle \mathcal{M}, \mathcal{G}, \mathcal{L} \rangle$, where

- $\mathcal{M}$ is the set of messages, regulated by $\mathcal{P}$. They are called $\mathcal{P}$-messages.
- $\mathcal{G}$ is a distributed group of *agents*, sometimes called a "policy-group," that are permitted to send and receive $\mathcal{P}$-messages, and thus are the participants in policy $\mathcal{P}$.
- $\mathcal{L}$ is the set of rules regulating the exchange of $\mathcal{P}$-messages between members of group $\mathcal{G}$, called the *law* of this policy. Broadly speaking, the law determines who in group $\mathcal{G}$ can send which $\mathcal{P}$-messages to whom, and what should the effect of such a message be.

For example, the components of the policy for secure-bid auction are as follows: the group $\mathcal{G}_{\mathcal{SB}}$ consists of all agents participating in the auction, including the auctioneers. The set of messages $\mathcal{M}_{\mathcal{SB}}$ consists of all the messages exchanged during the auction including: initiating an auction, issuing a bid, and announcing the results; and the law $\mathcal{L}_{\mathcal{SB}}$ is the set of rules described above for $\mathcal{SB}$, written in a given formal language. We introduce such a language in the following section.

It should be pointed out that we take a policy to have an independent existence, separate from the agents participating in it. We provide means for an agent to *join* a given policy $\mathcal{P}$—subject to the law of this policy—which will enable this agent to send and receive $\mathcal{P}$-messages.

3.2 The Law

A law $\mathcal{L}$ of a policy $\mathcal{P}$ determines the treatment of $\mathcal{P}$-messages is defined by specifying what should be done when such a message is sent, and when it arrives. More specifically, we deal with the following two kinds of events that are regulated under RI[2]:

- `sent(x,m,y)` — occurs when agent `x` sends an $\mathcal{L}$-message `m` addressed to `y`. The receiver `x` is considered the *home* of this event.
- `arrived(x,m,y)` — occurs when an $\mathcal{L}$-message `m` sent by `x` arrives at `y`. The receiver `y` is considered the *home* of this event. If the destination is the keyword `all`, `m` is multicasted to all members of the group. The sender `x` is considered the *home* of this event.

We assume no prior knowledge of, or control over, the occurrence of these *regulated events*. But the

[1] We will use the names RI and Moses somewhat interchangeably.

[2] Note that RI regulates some additional types of events, which are not relevant to security, and are, thus, ignored here.

effect that any given event e would actually have is prescribed by the law $\mathcal{L}$ of the message in question. This prescription, called the *ruling* of the law for this event, is a (possibly empty) sequence of *primitive operations* (discussed later) which are to be carried out at the home of e, as the immediate response to its occurrence.

Structurally, the law $\mathcal{L}$ is a pair $\langle \mathcal{R}, \mathcal{CS} \rangle$, where $\mathcal{R}$ is a fixed set of rules defined for the entire group $\mathcal{G}$ of the policy in question, and $\mathcal{CS}$ is a mutable set $\{\mathtt{CS}_x \mid x \in \mathcal{G}\}$ of what we call *control states*, one per member of the group. These two parts of the law are discussed in more detail below.

The control state $\mathtt{CS}_x$: This is the part of the law $\mathcal{L}$ that is associated with the individual member x of a group. It is a bag of terms, called the *attributes* of this member. The main role of these attributes is to enable $\mathcal{L}$ to distinguish between different kinds of members, so that the ruling of the law for a given event may depend on its home. Some of the attributes of an agent have a predefined semantics, such as the attribute self(n) where n represents the name of the member. However, the semantics of attributes for a given group is defined by the law. For instance, in our implementation of the Chinese Wall policy, Section 4.1, an analyst x might have an attribute companyPermit(c), which means that x is allowed to access company's c data.

The Primitive Operations: The operations that can be included in the ruling of the law for a given regulated event e, to be carried out at the home of this event, are called *primitive operations*. They are "primitive" in the sense that they can be performed *only* if thus authorized by the law. These operations include:

- Operations that change the CS of the home agent. Specifically, we can perform the following operations: (1) +t which adds the term t to the control state; (2) -t which removes the term t; (3) t1←t2 which change term t1 with term t2; and (4) incr(t(v),x) which increments the value v of a term t with some quantity x.
- Operation forward(x,m,y) emits to the network the message m addressed to y. (When a message thus forwarded to y arrives, it would trigger at y the event arrived(x,m,y).) The most common use of this operation is in a ruling for event sent(x,m,y), where operation forward (with no arguments) simply completes the passing of the intended message.
- Operation deliver(m) delivers the message m to the home-agent. The most common use of this operation is in a ruling for event arrived(x,m,y), where operation deliver (with no arguments) simply delivers the arriving message to the home agent.

The global set of rules $\mathcal{R}$: The function of $\mathcal{R}$ is to evaluate a ruling for any possible regulated-event that occurs at an agent with a given control-state. In our current model, $\mathcal{R}$ is represented by a very simple Prolog-like program—or, if you will, a set of situation–action rules. When this "program" $\mathcal{R}$ is presented with a goal e, representing a regulated event, and with the control-state of the home agent, it produces a list of primitive-operations representing the ruling of the law for this event. For the details of this formulation the reader is referred to [15]; here we will illustrate it with a detailed example in Section 3.4.

3.3 The Distributed Enforcement Mechanism

The law for a given policy $\mathcal{P}=\langle \mathcal{L}, \mathcal{M}, \mathcal{G} \rangle$ is enforced in principle as follows: there is a *controller* associated with each member of group $\mathcal{G}$, logically placed between the agent and the communications medium, as it is illustrated in Figure 1. All controllers have identical copies of the global set of rules $\mathcal{R}$ of $\mathcal{L}$, and each controller maintains the control states of the agents under its jurisdiction.

All controllers have identical copies of the global set of rules $\mathcal{R}$ of $\mathcal{L}$, and each controller maintains the control states of the agents under its jurisdiction. The steps taken when a member x wishes to send a $\mathcal{P}$-message m to a member y are:

1. x sends m to its assigned controller. The controller evaluates the ruling of the law $\mathcal{L}$ for the event send(x,m,y) and it carries out this ruling. If part of the ruling is to forward the message m to y, the controller sends m to the controller assigned to y.

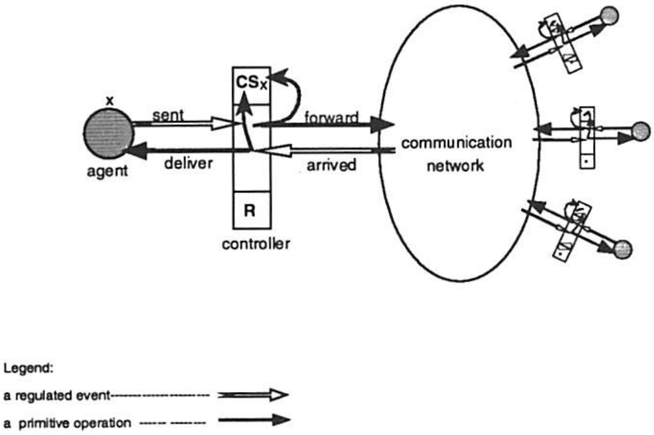


Figure 1: Enforcement of the Law

2. when `m` arrives to `y`'s controller it generates an `arrived(x,m,y)` event. The ruling for this event is computed and carried out. The message `m` is delivered to `y` if so required by the ruling.

The essential aspect of this architecture is that *all controllers have identical copies of the law.* It is in this sense that the law is said to be global to the group.

The correctness of the proposed mechanism is based on the following assumptions: (1) messages are securely transmitted over the network, and (2) $\mathcal{P}$-messages are sent and received only via correctly implemented controllers, interpreting law $\mathcal{L}$. To ensure the first of these conditions, every agent belonging to $\mathcal{G}$, and each controller, have a pair of (RSA) keys: a public key known to a trusted authority and a secret key known only by itself. If the messages sent across the network are digitally signed, their authenticity is guaranteed as long as the private key is not disclosed.

Condition (2) above is more problematic, and can be handled at two levels of security. First, if one is willing to trust the OS-kernel of the hosts of all members of the policy-group $\mathcal{G}$ — which may be the case within the intranet of a given enterprise – then this condition can be satisfied by placing the controllers in the OS kernels. Each controller would acquire the law that needs to be interpreted from some trusted authority.

One way to handle the case of untrusted OS-kernels is to ensure the integrity of the controllers by building them into *physically secure coprocessors* [22, 23], or into *smart cards* [11]. Such a secure device consists of a CPU, non-volatile memory, encryption hardware and special sensing circuitry to detect intrusion. The sensing circuitry erases non-volatile memory before attackers can penetrate far enough to disable the sensors or read memory contents. If Moses is implemented on such physically secure hardware devices, the receiver of a $\mathcal{P}$-regulated message has a high degree of trust that this message is authentic, in the sense that the message has been sent by a genuine controller interpreting law $\mathcal{L}$ of the policy $\mathcal{P}$.

Controllers can also be trusted if they are maintained as a public utility by a large, trustworthy, financial institution—like Visa or Master Card—by an Internet provider, or by the post offices of various countries. More about such controller utility can be found in a technical report obtainable from the authors.

3.4 Example: A Capability Based Regime

In centralized systems protection has been traditionally realized by means of capabilities and access-control lists, each of these models offering its advantages. In distributed systems however, the full power of capabilities has not been realized so far. In particular, in timesharing operating systems like Hydra [6] and StarOs [10] it was possible to control dissemination of capabilities by specifying whether a given capability can be moved or delegated to others. This feature is no longer supported in capability-based distributed systems [16], because nothing prevents a user from duplicating the capabilities he holds. We do not have this problem because capabilities are kept by controllers, which are *trusted* to execute only prescribed operations. This is demonstrated by policy $\mathcal{CR}$ which imposes a capability based access control regime in which *capabilities can be moved* from one agent to another but *cannot be copied*[3].

The components of policy $\mathcal{CR}$ are as follows: The group $\mathcal{G}_{\mathcal{CR}}$ consists of the servers and all their clients. The set $\mathcal{M}_{\mathcal{CR}}$ consists of all the messages exchanged between the server and its clients; and the law $\mathcal{L}_{\mathcal{CR}}$ mirroring the rules described above, is

[3] This is only a finger exercise, meant to illustrate the mechanism; a full implementation of capabilities should consider copyable capabilities, revocation, etc.

presented in Figure 2.

The set $\mathcal{R}_{\mathcal{CR}}$ of this law consists of four rules. Each is followed with a comment (in italic), which together with the following discussion, should be understandable even for a reader not familiar with Prolog. Consider a set of clients that perform operations by sending messages of the form `execute(o,op,p)` to servers, where `op` is an operation to be executed on object `o`, and `p` are parameters for this operation. We assume that initially clients have in their control state arbitrary sets of capabilities represented by terms of the form `capability(o,ar)`, where `o` is the name of an object, `ar` is the set of access rights the agent has for `o`. What gives these terms their meaning as access privileges is Rule $\mathcal{R}1$ of our law, is that a request to `execute` an operation `op` on some object `o` is forwarded only if the sender has an attribute `capability(o,ar)` *and* if `op` belongs to `ar`. When such a message arrives at a server, it is delivered by Rule $\mathcal{R}2$. Note that the servers need not know about our access control scheme, they just respond to every request they receive.

The rest of this law defines the manner in which capabilities are to be moved from one agent to another. This is done as follows: by rule $\mathcal{R}3$, if the owner of a capability for object `o`, sends a `move(capability(o,ar))` message, his capability for `o` is removed from his control state. By rule $\mathcal{R}4$, the arrival of a `move(capability(..))` message causes the addition of the corresponding capability in the control state of the receiver.

$\mathcal{I}$*nitially:* Every client has in its control state various attributes of the form `capability(O,AR)`, where `O` is an object on which the client has the set `AR` of access rights.

$\mathcal{R}1$.
```
sent(C,execute(Op,O,P),S) :-
      capability(O,AR)@CS,
      member(Op,AR),
      do(forward).
```

A request to `execute` *an operation* `Op` *on some object* `O` *is forwarded only if the sender has a capability for* `O` *and if* `Op` *belongs to* `AR`*, the set of access rights.*

$\mathcal{R}2$.
```
arrived(C,execute(Op,O,P),S) :-
      do(deliver).
```

Arrived `execute` *messages are delivered without further ado.*

$\mathcal{R}3$.
```
sent(C,move(capability(O,AR)),D) :-
      capability(O,AR)@CS,
      do(-capability(O,AR)),
      do(forward).
```

A `move(capability(..))` *will be forwarded only if the sender has a capability for the object* `O`*. The sender's capability is removed from his control state.*

$\mathcal{R}4$.
```
arrived(C,move(capability(O,AR)),D) :-
      do(+capability(O,AR)).
```

The arrival of a `move(capability(..))` *message causes the addition of the corresponding capability in the control state of the receiver .*

Figure 2: Law $\mathcal{L}_{\mathcal{CR}}$ - Establishing a capability based regime

3.5 The Creation and the Maintenance of a Policy

Under our current implementation of the Moses toolkit, a new policy $\mathcal{P}$ is established by creating a number of controllers that interpret law $\mathcal{L}_{\mathcal{P}}$ and a server that provides persistent storage for the law $\mathcal{L}$ of this policy– including the control-states of all members of the policy-group $\mathcal{G}$. This server is called the *secretary* of $\mathcal{P}$, to be denoted by $\mathcal{S}_{\mathcal{P}}$. The following are some of the services provided by such a secretary.

For a process `x` to be able to exchange $\mathcal{P}$-messages under a policy $\mathcal{P}$, it needs to send a `connect(a)` message to $\mathcal{S}_{\mathcal{P}}$, asking to be associated with some agent `a` that is a member of the group $\mathcal{G}$ of $\mathcal{P}$. $\mathcal{S}_{\mathcal{P}}$ is likely to require authentication, which can be in form of a password, an X.509 certificate [12] or the recently developed SDSI certificate [17]. If the secretary agrees to make the connection, it would assign `x` to some controller interpreting law $\mathcal{L}_{\mathcal{P}}$, after providing this controller with the current control-state and the public key of the agent.

Once this connection is made, the interaction of `x` with the various members of policy $\mathcal{P}$ does not directly involve the secretary $\mathcal{S}_{\mathcal{P}}$. However, if some event at `x` ends up changing the control-state of the member `a` it is associated with, this change would be automatically communicated to $\mathcal{S}_{\mathcal{P}}$.

The secretary of a policy also acts as a name server for the members of its group $\mathcal{G}$, and it provides means for admitting new members into $\mathcal{G}$, and for removing existing members from it. These operations, which are subject to $\mathcal{L}$, are not discussed here in detail.

Finally, we note that a policy does not have to be supported by a single secretary. It is possible, in principle, for a policy to have several secretaries, each maintaining a subgroup of $\mathcal{G}$.

3.6 Fault Tolerance and Scalability

Regulated interaction lends itself to fault tolerant and scalable implementations, as we argue briefly below.

Fault Tolerance: Since RI assumes nothing about the interacting agents, it is tolerant to all their failures, even of a Byzantine kind. But RI is sensitive to two kinds of failures: (a) the failure of the `secretary`, which may have a devastating effect on the long term existence of the policy-group, even if it has no effect on the immediate interaction between its members; and (b) the failure of a controller. Fail-stop failures of these two kinds can be handled by well known methods. Failures of the secretary can be addressed by means of the state-machine approach [19], using a toolkit such as Isis [2] for the active replication of the secretary. Failures by controllers can be analogously handled by replication of each controller. Alternatively, given a reliable secretary, it may be sufficient for the controllers to notify the secretary of all state changes.

Scalability: Since the law is enforced strictly locally, by the controller of each agent, the size of the policy-group has no effect on the interaction between its members. Therefore, RI is *naturally scalable*, particularly in the case of an *open group.* However, when a group is supported by a single secretary, as in our current implementation, then the size of the group does affect operations such as finding the name of a fellow member of a group, or reporting to the secretary a change of the CS of a given member. But this has a second order effect on the efficiency of interaction under RI.

3.7 Implementation Status

An experimental prototype of the Moses toolkit has been implemented. Our controllers are written in Java, so that Moses toolkit is portable to different platforms. Because our rules do not require the full power of Prolog language, we have built an interpreter for the needed subset of Prolog. This implementation distinguishes between two types of agents:

(i) A *bounded agents*, driven by a specific program (which is what "binds" it). This programs, which can be written in C, C++, Java, or Prolog, uses a set of preprogrammed primitives for communication with Moses' controllers.

(ii) An *unbounded agents*, which represents a human, not bound by any program. Such an agent communicates with its assigned controller via Netscape, using application specific interfaces consisting of HTML documents with embedded applets. Our choice was motivated by (1) the almost universal deployment of WWW browsers; and (2) the ease of learning to use this interface.

The implementation has been tested on UNIX platforms including SunOS and Solaris. The controllers have not yet been deployed on physically secure devices.

4 Implementation of Our Example Policies

To illustrate the expressive power of the proposed mechanism, we present here the implementation in Moses of two disparate policies mentioned previously: the Chinese Wall policy, and the sealed-bid auction policy. Recall that although these policies, and the one discussed in Section 3.4, are defined by separate laws, unrelated to each other, a given agent may be subject to several of these policies, with respect to different modes of interaction it is involved in.

It should be pointed out that our implementation of the sealed-bid policy assumes no loss of messages and no failure of controllers. On the other hand the implementation of the Chinese Wall policy is robust with respect to loss of messages and fail-stop failures of controllers.

4.1 An Implementation of the Chinese Wall Policy

This policy is established by law $\mathcal{L}_{CW}$, displayed in Figure 3. We assume that the servers of the distributed financial database are trusted to respond to $\mathcal{L}_{CW}$ messages of the form `request(c)`, where `c` identifies a company, by means of `respond(c,data)`, where `data` represents information about `c`. Note that under this law no explicit access control is required on the part of the server.

Under law $\mathcal{L}_{CW}$ a client is authorized to access data belonging to a company `c` if either of the following conditions is satisfied:

1. the client has a clique permit for clique `q` to which `c` belongs—such a permit is represented by a term `cliquePermit(q)` in the control state of the client;
2. the client has a company permit for company `c`—represented by the term `companyPermit(c)` in the control state of the client.

If the access is granted based on a `cliquePermit`, then this permit is automatically removed to prevent the client from accessing data regarding a competitor company, and replaced with a `companyPermit` for company `c`.

This implementation deals with the exceptional situation when a first time request for a company is not honored for whatever reason. In this case, a client should be able to access information about another company belonging to the same competition clique. That is why we chose to replace a `cliquePermit` by a `companyPermit` at the time the client *receives* the information and not when he makes the *request*. In this respect, the protocol presented is tolerant to server faults of type fail/stop and to lost messages. Note also that Rule $\mathcal{R}1$ checks for the presence of a company permit or a clique permit at the time a request is sent. This check is not needed for the correctness of the protocol, it is performed to ensure that only potentially valid requests are sent to servers and thus diminish the possibility of server congestion, and thus of denial of service.

Initially: Every client has in its control state some attributes of the form `cliquePermit(Q)`

$\mathcal{R}1$.
```
sent(U,request(C),S) :-
     (companyPermit(C)@CS |
     (belongsTo(C,Q),cliquePermit(Q)@CS)),
     do(forward).
```

If a user `U` *has a permit for company* `C` *or he has a permit for clique* `Q` *to which* `C` *belongs, then the request is forwarded.*

$\mathcal{R}2$.
```
arrived(U,request(C),S) :-
   do(deliver),
   do(+requested(C,U)).
```

If a `request` *message arrives at a server, the message is delivered. Also, a term* `requested(C,U)` *is added to the control state to record the fact that user* `U` *is requesting data about company* `C`.

$\mathcal{R}3$.
```
sent(S,response(C,Data),U) :-
     requested(C,U)@CS,
     do(-requested(C,U)), do(forward).
```

In response to a `request(C)` *message, a server can send a* `response(C,Data)`. *Note that this message may be sent only by the server to whom the user addressed the request, i.e. the server which has the term* `requested(C,U)`.

$\mathcal{R}4$.
```
arrived(S,response(C,Data),U) :-
     companyPermit(C)@CS,do(deliver).
```

If user `U` *receives information about a company* `C`, *for which he has a permit, the data is delivered.*

$\mathcal{R}5$.
```
arrived(S,response(C,Data),U) :-
     belongsTo(C,Q),cliquePermit(Q)@CS,
     do(-cliquePermit(Q)),
     do(+companyPermit(C)),
     do(deliver).
```

If user `U` *receives information about a company* `C`, *belonging to a clique* `Q` *he loses the permit for clique* `Q` *and gets a permit for the company* `C`.

$\mathcal{R}6$. `belongsTo(att,communication).`

$\mathcal{R}7$. `belongsTo(ibm,communication).`

⋮

This rules state that att and ibm belongs to communication competition clique. There will be one such rule for every company whose financial information is available.

Figure 3: Law $\mathcal{L}_{CW}$ for Chinese Wall Policy

4.2 An Implementation of the Sealed-Bid Auction Policy

We introduce $\mathcal{L}_{\mathcal{SB}}$, displayed in Figure 4, which implements the law of the sealed-bid auction policy $\mathcal{SB}$ introduced in Section 2.2. This law regulates two different types of messages: the messages that can be used by the agents involved in this policy to withdraw and deposit money, and the messages related to the auction per se.

The deposit and withdrawal of money. We assume that an agent called bank is a financial institution to which both auctioneers and bidding agents have accounts. The bank is trusted to respond only to $\mathcal{P}_{\mathcal{SB}}$ messages of the form `transaction(..)` and we assume it performs the financial operations correctly. Any agent has in its control state a term `cash(amount)` where `amount` is the sum available for bidding. Under this law, agents are allowed to transfer money from their account to their bidding fund and vice versa. An agent, wishing to withdraw sum `s` from its bank account sends to `bank` the message `transaction(type(withdrawal),sum(s))`. The `bank` responds with an `addCash(s)` message if the transaction is valid. When the agent receives such a message the amount `s` is automatically added to its cash (Rule $\mathcal{R}4$). Similarly, an agent can make a deposit in amount of `s` into his bank account, by sending the message `transaction(type(deposit),sum(s))` (Rule $\mathcal{R}1$). This message will be forwarded to the `bank`, only if the agent has enough cash, after his `cash` term is decreased accordingly (Rule $\mathcal{R}1$).

The auction process. Intuitively, a sealed-bid auction proceeds as follows. First, an auctioneer `x` can start a sale by multicasting the message `startAuction(item(i),end(et))`, where `i` is a unique description of the item to be sold, and `et` is the time when the auction ends (Rule $\mathcal{R}5$). At the same time, a term `auction` is added to the control state of the auctioneer which records the highest bid made so far (initially 0) and the name of its issuer (initially null). When such a message arrives at an agent `y`, a term `bided(x,i,et,0)` is added to `y`'s control state (Rule $\mathcal{R}6$). This term serves two purposes: (i) to enable `y` to bid for the item as many times as he wants to, but only in the allotted time; and (ii) to record the maximal bid made by `y` for item `i` (initially 0). An agent `y` makes a bid of `val` dollars for item `i` by sending a message `bid(i,val)` to the auctioneer. Such a message is forwarded to the destination only if the following conditions are met: (i) the deadline `et` has not yet passed, (ii) `val` is the highest bid `y` made so far, and (iii) `y` has enough cash (Rule $\mathcal{R}7$). Also, the cash amount `y` possesses is decreased, and the `bided` term is modified to reflect that `val` is his highest bid for `i`.

All such `bid` messages arrive at the controller of the auctioneer `x`, which maintains a term `auction` recording the largest bid so far and the name of the winner (Rule $\mathcal{R}8$). Note that the auctioneer himself never gets the bids, the winner is determined by the controller automatically, thus ensuring bidders privacy and correctness of the computation.

The auction of item `i` is finalized by a message `endAuction(i)` from the auctioneer, which he is allowed to send only after the deadline has passed ($\mathcal{R}9$). The effects of this message are described briefly below. First, the auctioneer's amount of cash is increased by the value of the highest bid, which he can later deposit in the bank, by Rule $\mathcal{R}1$. Second, the controller of the auctioneer will multicast the message `endAuction(i,w)` containing the identifier of the winner to all group members. Note that the auctioneer himself does not know the identity of the winner—only his controller has this information. When the message `endAuction(i,w)` arrives at a losing bidder, his cash amount is increased by the highest value he bided on `i`, so he does not loose any money (Rule $\mathcal{R}10$). When this message arrives at the winner `w`, then, by the same rule, the message is delivered, thus notifying `w` that he won.

Note that for the sake of simplicity, we do not address here the situation of an auctioneer denying the winner his prize. This can be prevented by having the controller of the auctioneer in question send the winner an appropriate certificate.

Initially: Every agent has in his control state an attribute `cash(Amount)` where `Amount` is the sum the agent can use for bidding (initially 0).

$\mathcal{R}1$.
```
sent(X,transaction(type(T), sum(S)),bank) :- (T = withdrawal |
          (T=deposit, cash(Amount)@CS, S<Amount, do(dcr(cash(Amount),S)))), do(forward).
```

If the transaction is a `deposit`*, the message is forwarded only if the client has enough cash. If this is the case, the amount of cash is decreased by* `S`.

$\mathcal{R}2$.
```
arrived(_,transaction(type(T), sum(S)),bank) :- do(deliver).
```

$\mathcal{R}3$.
```
sent(bank, addCash(S), _) :- do(forward).
```

Messages sent to the bank are delivered, and messages sent by the bank are forwarded without further ado.

$\mathcal{R}4$.
```
arrived(bank, addCash(S), X) :- cash(Amount)@CS, do(incr(cash(Amount),S)).
```

The cash amount is increased by `S`*, when a successful bank transfer is performed.*

$\mathcal{R}5$.
```
sent(X,startAuction(item(I),end(ET)),all) :- auctioneer@CS,
          not (auction(I,_,_,_)@CS), do(+auction(I,ET,null,0)),
          do(forward).
```

An auctioneer can start an electronic auction by sending a `startAuction` *message to* `all` *members. The message contains an identifier* `I` *of the item to be auctioned and the time* `ET` *when the auction ends.*

$\mathcal{R}6$.
```
arrived(X,startAuction(item(I),end(ET)),Y) :-
          do(+bided(X, I, ET, 0)),do(deliver).
```

A `startAuction` *message is delivered to the destination. A term* `bided(X, I, ET, 0)` *is added to the control state of the receiver.*

$\mathcal{R}7$.
```
sent(Y,bid(I, Val),X) :- not(auctioneer@CS),clock(T), T < ET,
          bided(X, I, ET, V)@CS, V<= Val, cash(Amount)@CS,
          V + Amount >= Val, do(bided(X, I, ET, V)←bided(X, I, ET, Val)),
          do(dcr(cash(Amount)), Val-V), do(forward).
```

If the time to bid has not expired and the bidder has enough cash, then a `bid` *message containing* `Val` *the value of the bid, is forwarded to the initiator.*

$\mathcal{R}8$.
```
arrived(Y,bid( I, Val),X) :- auction(I, ET, W, Max)@CS, Val > Max,
          do(auction(I,ET,W,Max)←auction(I,ET,Y,Val)).
```

If `Val` *is the biggest amount bided so far for* `I`*, the sender's identifier is recorded in the* `auction` *term along with* `Val`.

$\mathcal{R}9$.
```
sent(X,endAuction(I),all) :- clock(T),T > ET +100,
          cash(Amount)@CS,auction(I,ET,W,Max)@CS,
          do(incr(cash(Amount),Max)), do(forward(X,endAuction(I, W),all)).
```

Only the auctioneer `X` *who organized the sale for item* `I` *can send a* `endAuction` *message. The identifier of the winner* `W` *are sent to all group members. Also, the money collected from the winner are added to* `X`*'s cash.*

$\mathcal{R}10$.
```
arrived(X,endAuction(I,W),Y) :- cash(Amount)@CS,
          bided(X, I, ET, Val)@CS, do(-bided(X,I, ET, Val)),
          W=Y→do(deliver) | do(incr(cash(Amount),Val)).
```

When a message `endAuction` *arrives at a bidder* `Y`*, if he is a loser he gets his money back.*

Figure 4: Law $\mathcal{L}_{SB}$ for Sealed-Bid Auction Policy

5 Related Work

The need for a mechanism for specifying security policies as an alternative to hard coding them into an application occurred to several researchers. Theimer, Nichols and Terry [21] introduced a concept of *generalized capabilities*. Such capabilities contain access control programs (ACP) encoding the security policy to be enforced with respect to this capability. When a server receives a request accompanied by such a generalized capability, it executes the ACP to determine whether the request is valid or not.

Finally, Blaze, Feigenbaum and Lacy [3] built a toolkit called PolicyMaker which can interpret security policies. An agent receiving a request gives it for evaluation to PolicyMaker together with its specific policy, and the requester's credentials. On this basis the request can be found to be valid, invalid or trust can be deferred to third parties. One of the main differences between this work and ours is that PolicyMaker provides no enforcement. In particular, after asking PolicyMaker for its ruling one can proceed by ignoring it.

Also, in both these approaches the rights a user has are *static*: they cannot be modified in accordance with its actions. Thus, a large range of security policies, like separation of duties [5], Chinese Wall, and the movable but uncopyable capabilities, where the *state* of a user determines his rights, cannot be enforced.

6 Conclusion

The essence of the security mechanism proposed here is the existence of a law that is guaranteed to be observed by *all* members of a given policy-group. It is this uniform[4] law that allows the members of the group governed by it to trust each other. This *distributed trust* has several beneficial consequences: (a) it simplifies the formulation of a wide range of policies, some of which cannot be supported by traditional means; (b) it allows a single agent to operate under several distinct policies; (c) it makes the enforcement of policies more efficient; and (d) it makes the mechanism itself scalable. This trust relies on the integrity of the controllers, and on the ability to correctly identify $\mathcal{P}$-messages. These conditions can be met, with a very high level of confidence, by implementing controllers on physically secure devices, and by appropriate authentication protocols. In some cases, however, it should be sufficient to build the controllers into the kernel of the operating systems involved.

[4] The law is uniform with respect to the members of each group.

References

[1] J. R. Anderson. A security policy model for clinical information systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.

[2] K.P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[3] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust managemnt. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.

[4] D. Brewer and M. Nash. The Chinese Wall security policy. In *Proceedings of the IEEE Symposium in Security and Privacy*. IEEE Computer Society, 1989.

[5] D.D. Clark and D.R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium in Security and Privacy*, pages 184–194. IEEE Computer Society, 1987.

[6] E. Cohen and D. Jefferson. Protection in the HYDRA operating system. In *Operating Systems Principles*, pages 141–160. ACM, Nov. 1975.

[7] S. Foley. The specification and implementation of 'commercial' security requirements including dynamic segregation of duties. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, April 1997. (to appear).

[8] S Foley, L. Gong, and X. Qian. A security model of dynamic labelling providing a tiered approach to verification. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.

[9] M. Franklin and M. Reiter. The design and implementation of a secure auction service. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 2–14, May 1995.

[10] A. Jones, R. Chansler Jr., I. Durham, K. Schwans, and S. Vegdahl. StarOS, a multiprocessor operating system for the support of task forces. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 117–127, Dec 1979.

[11] M. Jones and B. Schneier. Securing the World Wide Web: Smart Tokens and their implementation. In *Proceedings of the Fourth International World Wide Web Conference*, pages 397–409, December 1995.

[12] S. Kent. Internet privacy enhanced mail. *Communications of the ACM*, August 1993.

[13] N.H. Minsky. The imposition of protocols over open distributed systems. *IEEE Transactions on Software Engineering*, February 1991.

[14] N.H. Minsky and V. Ungureanu. Regulated coordination in open distributed systems. In *Proc. of Coordination'97: Second International Conference on Coordination Models and Languages; LNCS 1282*, pages 81–98, September 1997.

[15] N.H. Minsky, V. Ungureanu, W. Wang, and J. Zhang. Building reconfiguration primitives into the law of a system. In *Proc. of the Third International Conference on Configurable Distributed Systems (ICCDS'96)*, March 1996. (available through `http://www.cs.rutgers.edu/~minsky/`).

[16] S. Mullender, G. Rossum, A. Tanembaum, R. Van Renesse, and H. Staveren. Amoeba:a distributed operating system for the 1990s. *IEEE Computer*, May 1990.

[17] R. Rivest and B. Lampson. SDSI-a simple distributed security infrastructure. Technical report, 1996. http://theory.lcs.mitedu/~rivest/sdsi.ps.

[18] Ravi Sandhu. Lattice-based access control models. *IEEE Computer*, November 1993.

[19] F.B. Schneider. Implementing fault tolerant services using the state machine approach. *ACM Computing Surveys*, 22(4):300–319, 1990.

[20] A. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, 1995.

[21] M. Theimer, D. Nichols, and Douglas Terry. Delegation through access control programs. In *Proceedings of Distributed Computing System*, pages 529–536, 1992.

[22] J.D. Tygar and B. Yee. Dyad: A system for using physically secure coprocessors. Technical Report CMU-CS-91-140R, CMU, 1991.

[23] B. Yee. *Using Secure Coprocessors.* PhD thesis, Carnegie Mellon University, May 1994.

Operating System Protection for Fine-Grained Programs

Trent Jaeger Jochen Liedtke Nayeem Islam
IBM T. J. Watson Research Center
Hawthorne, NY 10532
Emails:{jaegert|jochen|nayeem@watson.ibm.com}

Abstract

We present an operating system-level security model for controlling fine-grained programs, such as downloaded executable content, and compare this security model's implementation to that of language-based security models. Language-based security has well-known limitations, such as the lack of complete mediation (e.g., for compiled programs or race condition attacks) and faulty self-protection (effective security is unproven). Operating system-level models are capable of complete mediation and self-protection, but some researchers argue that operating system-level security models are unlikely to supplant such language-based models because they lack portability and performance. In this paper, we detail an operating system-level security model built on the Lava Nucleus, a minimal, fast μ-kernel operating system. We show how it can enforce security requirements for fine-grained programs and show that its performance overhead (with the additional security) can be virtually negligible when compared to language-based models. Given the sufficient performance and security, the portability issue should become moot because other vendors will have to meet the higher security and performance expectations of their customers.

1 Introduction

We demonstrate how operating system protection can be used to control fine-grained programs flexibly and efficiently. Operating systems use hardware-based protection to isolate processes from one another. However, the way that current operating systems implement this protection has caused researchers to deem them too slow and inflexible for controlling fine-grained programs. Fine-grained programs have different protection domains and may interact often in the course of a computation. The effect of a large number of protection domain crossings must be handled securely (i.e., correctly with respect to the security requirements) to prevent attacks and efficiently to minimize performance degradation. In this paper, we show that a security model implemented on a fast and flexible IPC mechanism can enforce security requirements that language-based systems cannot with little performance impact.

Several operating systems use hardware-based protection to prevent processes from inadvertently and/or maliciously modifying one another. Each process has an address space that defines a set of memory segments and the process's access rights to those segments. A process can only access memory in its own address space. In addition, the operating system has a security model that associates processes with their access rights to system resources. Using address spaces of suitable granularity and process access rights to controlled resources, an operating system can control a process's operations as desired.

However, operating system security models have been deemed to lack the performance and flexibility necessary to control fine-grained programs. While some systems have been built that efficiently control processes in dynamically-defined protection domains [3, 8, 14], these systems have been applied only to more traditional applications (e.g., PostScript interpreters). In an application composed of fine-grained programs, programs with different protection domains interact often (perhaps as much as on each method invocation). In a recent paper, Wallach *et al.* [25] discard address space-based protection from consideration for applications with fine-grained programs by noting that IPC between two COM objects on Windows NT takes 1000 times longer than a procedure call (230 μs to 0.2 μs). We claim that this discrepancy can be virtually eliminated while gaining security and maintaining flexibility.

We have developed a prototype implementation of a flexible security model for controlling downloaded content. This model is implemented on the Lava Nucleus. The Lava Nucleus provides address spaces, threads, fast IPC, flexible paging, and IPC interception that enable efficient and flexible control of processes. In this paper, we primarily concentrate on the effectiveness of the Lava nucleus for implementing a flexible security model and its resultant performance. We show that fast IPC and IPC interception enable the implementation of *dynamically authorized* IPC that can be performed in as little as 9.5 μs. We believe further room for optimziation is possible, given an ideal estimate for dynamically authorized IPC is about 4 μs. Also, flexible page mapping in the Lava Nucleus enables objects of size greater than the hardware page size to be shared among processes, so coarse-grained sharing of memory between processes is possible.

The structure of this paper is as follows. In Section 2, we compare language-based and operating system-based security models. In Section 3, we describe an operating system security model for downloaded executable content. In Section 4, we describe the implementation of this model on the Lava Nucleus. In Section 5, we examine the performance of the prototype implementation and compare its performance to language-based models for fine-grained programs of the sort discussed by Wallach *et al.* In Section 6, we conclude and present future work.

2 Language vs. O/S Protection

The basic problem is to implement a security kernel that effectively and efficiently enforces the security requirements of downloaded executable content. The basic requirements of any security infrastructure are that it can[1] (adapted from [2]):

- assign permissions dynamically to individual content,
- mediate all controlled operations using such permissions (i.e., an operation that a process is not unconditionally trusted to invoke),
- manage the evolution of permissions as content is executed,
- protect itself from tampering

For a system that uses dynamically downloaded executable content, permissions must be assignable to individual content. The security kernel must be able to mediate all controlled operations. To enforce least privilege on content permissions may be based on application state and evolve as application state evolves. The security kernel must be able to control this evolution within reasonable limits. Lastly, the kernel must be able to protect itself from modifications that may result in tampering with its behavior.

A question that has re-appeared recently is whether language-based or operating system-based protection is better suited for effectively and efficiently controlling such fine-grained programs. Or otherwise stated: to what extent can operating system protection *efficiently* provide *effective* security and to what extent can language protection *effectively* provide *efficient* security? As described below, operating system protection has several advantages over language protection from a security perspective, but the cost of domain crossings make it questionable whether efficient operating system protection for fine-grained processes is possible. On the other hand, language protection can be implemented efficiently, but some key security safeguards are weakened such that effective security may be lost.

Traditionally, operating systems have enforced system security requirements because hardware-based protection provides significant advantages in the key areas of economy of mechanism, fail-safe defaults, and complete mediation [23]. The operating system's TCB can protect processes by restricting them to their own address spaces which can be enforced by a simple mechanism (at least compared to a compiler). Since only the program requested is placed in the address space, other, independent programs are not affected by its failure (assuming the operating system adequately protects itself from such failures). Also, since the operating system can intercept any interprocess communication (IPC) between processes, controlled operations by a program (including compiled ones) can be completely mediated by the operating system.

Language-based protection has gained favor in recent years, however. We attribute this popularity to three factors: (1) improvements in the development of "safe" languages; (2) the perception that programs will become increasingly fine-grained, and fine-grained domains are prohibitively expensive to enforce; and (3) lack of flexibility in operating system security models. "Type-safe" languages are strongly-typed (i.e., all data is typed and casting is restricted or prohibited) and do not permit direct addressing of the system's memory (i.e., no pointers). Therefore, all data is accessed according to its interface, so complete mediation of controlled operations in pro-

[1] There is an additional requirement that any "security kernel" be simple enough that independent evaluators can assess whether it will operate properly. While we will not prove such a feature about our system here we do attempt to keep the security model as simple as is feasible.

grams written in such languages is possible. Other "safe" languages, such as Safe-Tcl [4, 21], Penguin [7], and Grail [24] depend on removal of operations that would enable the language security infrastructure to be circumvented. Also, with the increased popularity of component-based system infrastructure, such as Java Beans, and the ability to downloaded code dynamically, are leading people to predict that programs with become more fine-grained. Mediation of IPC between processes has significant performance implications for operating systems because they may need to perform expensive operations, such as handling TLB misses, upon domain changes. Also, the security models of current commercial operating systems, such as UNIX and DOS/Windows, lack the flexibility to dynamically assign permissions to user processes or permit controlled sharing of memory among processes.

Language-based protection has some significant weaknesses, however. First, the TCB of a system depending on language protection is larger because now we must trust the compilers and code verifiers as well as the "system" TCB objects provided by the system. While compilers are much better understood than they once were, a minimal TCB is still preferred. Second, since all programs run within a single address space, fail-safety is not what it is in operating systems. This means that a security flaw will result in the attacker gaining all privileges that the virtual machine has. Given that single domain operating systems based on language protection are being built (e.g., Java OS), this means that compromise of the virtual machine can result in significant losses. Third, it is not clear what the actual size and number of components that can share a protection domains will be. In Wallach *et al.*, they measured 30,000 domain crossings per second. It is not explicit in their paper, but we assume that many of these domain crossings involved trusted classes whose use have few security implications. Therefore, we believe that many of these claimed domain crossings can be avoided. Lastly, and most obviously, language-based protection is language-specific and does not apply to compiled code. Therefore, complete mediation depends on a homogeneous system which we believe is unlikely.

Also, language-based protection has its own performance problems and the optimizations to improve performance introduce subtle security flaws. For example, in the JDK 1.2 specification [10], excess authorizations (on every method invocation since there may be many real domains within a single protection domain) are prevented by using the call stack to determine the current authorization context. However, the current call stack may not represent the actual context since classes that are no longer on the stack may influence the execution. In addition, security depends on the proper placement of calls to the authorization object. For applications, this means that code must be changed to control a previously uncontrolled object.

The question is whether operating system protection can be made flexible enough and efficient enough for fine-grained programs. We believe that the flexibility question has been answered for a long time, but the systems for which it was answered are no longer in wide use. Several research operating systems were developed that used capabilities to attach flexible protection domains to processes in a secure manner [16, 20, 28]. For example, Hydra [28] obtained flexibility by attaching capabilities to procedures. SCAP [16] and PSOS [20] demonstrated that a number of security properties could be enforced by these systems. However, the applications of the time had much simpler security requirements, so much less secure systems were adopted as de facto standards.

We, therefore, believe that the key problem to using operating system protection for fine-grained programs is performance. At IBM, we are developing the Lava Nucleus which is a minimal, fast, and flexible μ-kernel. It provides the basic system constructs, such as processes, address spaces, and threads, and provides general mechanisms for using these primitives, such as flexible memory mapping primitives, fast IPC, and IPC redirection. In this paper, we show that a flexible operating system protection model for downloaded executable content can be designed and that it can be implemented efficiently using the Lava Nucleus. From this, we measure the currently achievable performance of operating system protection and evaluate how this affects the overall system performance. In the future, we expect that language-based and operating system protection to be jointly used (depending on the security requirements) with operating system protection providing a simple, fail-safe security mechanism that may completely mediate controlled domains.

3 Security Model

An effective security model for downloaded executable content requires significant flexibility in the creation of principals, assignment of their permissions, and management of their permissions throughout the content's execution. We use Lampson's protection matrix [17] to help describe these requirements. It shows the relationships among subjects, objects, and the operations that subjects can perform on objects (permissions). Now consider the security requirements for controlling downloaded executable content described below.

- **Subjects:** There may be one subject per downloaded content, although some subjects may be reused within a session.

- **Objects:** The objects may refer to logical objects that must be mapped to the individual downloading principals system at runtime.

- **Permissions:** The set of permissions that a subject has at any time to objects may evolve as system's content (subjects) executes.

In traditional systems, the set of subjects is generally mapped to the set of users and a set of well-known services. This granularity is too coarse for downloaded content. Each content may be associated with a different principal that is an aggregate of basic principals (e.g., downloading principal executing content from a provider within a specific instance of an application). Also, in traditional systems, the set of objects is well-known. This may not be the case in downloaded content systems because the content providers may have limited knowledge about the systems upon which their content is run. In addition, the amount of policy specification can be reduced if the logical objects that are mapped to a specific principal's domain at runtime can be used. Lastly, enforcement of least privilege on content is more important than for traditional programs because these programs are transient and from only partially trusted sources (i.e., may have bugs). In traditional systems, permissions are assigned to principals and must cover every execution of that principal, or else some application executions would fail. For content, the state of an application may also be used to limit the permissions to those necessary for the task at hand.

To solve these problems, we propose an security model that provides basic security mechanisms and policy representations to enable the control of downloaded executable content. The model is shown in Figure 1. In this security model, a *secure booting mechanism* loads a nucleus of an operating system. The *nucleus* provides the basic functionality to create and execute processes and enables them to intercommunicate (via IPC). The nucleus initiates a *process load server* that creates subsequent processes and can dynamically load libraries and code components into existing processes. The process load server uses an *authentication server* to authenticate content and determine the content principal. A *derivation server* derives the permissions for this content principal. The process load server assigns a monitor to enforce a process's permissions. *Monitors* intercept and authorize all IPCs emanating from or directed to the controlled process. Monitors maintain a representation of the controlled processes access rights and can both add to and revoke access rights from the controlled process.

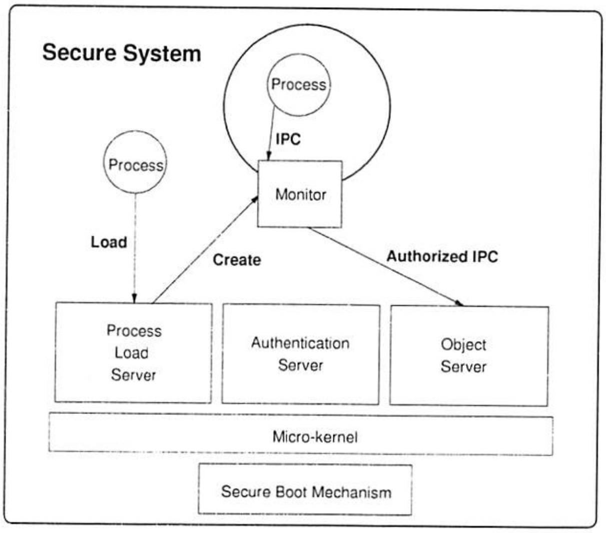


Figure 1: Security Model Architecture

In the design of our security model, we make the following assumptions. System administrators are completely trusted to set system policy. A set of certification authorities are trusted to vouch for the public keys of principals. A secure booting mechanism is trusted to initialize the operating system properly. We assume the existence of such a mechanism as proposed for the Taos operating system [27]. The kernel itself is trusted to create processes and threads properly, separate process address spaces, identify the source of IPC, and redirect the IPC of controlled processes to monitors. The authentication server is trusted to perform cryptographic operations correctly. The process load server is trusted to setup the processes and monitors properly, so the security requirements as specified can be enforced. Monitors are trusted within a domain limited by the rights that they can delegate to processes. Monitors may also be trusted to read and not leak server data to other processes. The system has limited trust in users, applications, and downloaded content.

The following three subsections describe how the first three of the security requirements of the system are enforced. The fourth security requirement's must be enforced throughout each of these operations.

3.1 Process Loading

The process load server receives load requests for objects and retrieves, validates, and loads the object. Some objects may be executable and some may not, but our discussion focuses on the loading of executable objects.

The process load server solves the following problems:

- retrieve requested executable content,
- uses the authentication server to verify the authenticity of content and derive the content principal,
- uses the derivation server to derive the content's permissions,
- find the process in which to load the content,
- load the content into that process

Our effort here is concentrated on a mechanism for loading executable content. We propose mechanisms and policy representations for authentication and permission derivation elsewhere [12, 13]. These mechanism enable permissions to be derived dynamically for downloaded content given limited input from multiple principals.

While IPC can be faster than 230 μs, it is still well understood that procedure calls are faster. IPC in the Lava Nucleus takes 4 times longer on a Pentium than a procedure call than Wallach *et al.* [26] measure for a PC. In addition, there are indirect costs that are a result of the context switch, such as the handling TLB and cache misses. While the Lava Nucleus is designed to enable these costs to be reduced, the fewer context switches the better.

To reduce these overheads, we want the ability to link supporting content in the requesting process. However, only links that ensure that both the requesting process and the downloaded content do not obtain any unauthorized access rights can be permitted. This is only possible if: (1) neither the downloaded content nor the requesting process gain any permissions as a result of the co-location; (2) the downloaded content is permitted access to the requesting process's data and vice versa; and (3) the downloaded content can run properly with the rights of the requesting process. For the first condition to hold, the permissions of the joint process must be the intersection of the permissions of the content loaded into it. Thus, neither process can use a permission unless both had it previously. Also, neither content may have data in its address space that it must keep secret from the other. In addition, the content and process must also be able to effectively perform their jobs with the resultant rights for the co-location to be feasible.

While these restrictions limit the content that can be loaded into the requesting process, a variety of useful content can still be downloaded and co-located. Trusted libraries can be co-located with the requesting process in many instances. For example, many Java classes in java.lang package (although not the Java ClassLoader whose functionality we are superseding) can be loaded into a requesting process. For example, the String class does not provide the user's process any additional rights (although it may be used to circumvent language-based security), so it can be loaded into the requestor's address space. Also, we think that all the classes in the java.io can be loaded into a requesting process, because restricted access to the file server can be enforced by the monitor. Of the 30,000 domain crossings per second measured by Wallach *et al.*, we expect that many of those do not really require a change in domain for the requesting process. Our experience with the FlexxGuard prototype system (a controlled Java interpreter) was that restricting the permissions of the Java system classes to that of the current applet being run still permitted many useful applications to be implemented [1].

3.2 Permission Management

A process's monitor also manages the evolution of its permissions throughout its execution. In general, permission changes occur for two reasons: (1) an application state change may warrant a change in the controlled process's permissions and (2) one content using another may result in a restriction of permissions to prevent information leakage. In the first case, a user may perform an operation that results in the delegation of rights to downloaded content. For example, the loading of a file into an application may result in the delegation of the permissions to read and write the file to content used in the application. These permission changes must be limited to prevent a process from obtaining an unauthorized right. Also, the interaction of two untrusted content processes may require that their permissions be intersected to prevent the callee from performing unauthorized operations on behalf of the caller.

Permission management requires:

- the ability to add rights to the current permissions,
- the ability to revoke rights from the current permissions,

- a mapping of events to permission transformations,

- a limit to the rights that can be delegated to content,

In this security model, a monitor can add a right to a process's permissions if: (1) the delegating process has the permission; (2) the delegating process is permitted to delegate that permission to the delegatee; and (3) the permission is within the *maximal permissions* for the process. Each monitor maintains a mapping of its processes to its delegating principals and permissions (called *assignment limits*) that that principal can delegate to this process. Any delegated right must be within the process's maximal permissions (both the initial current permissions and maximal permissions for a process are derived by the derivation service). This limits the rights available to a process in general.

Revocation of rights is more difficult because capabilities can be copied. To prevent a process from using a revoked capability, the monitor does not pass capabilities to its controlled processes. Instead, a descriptor is returned to the process which enables it to refer to the capability. Revocation of the capability invalidates the descriptor, so capabilities can be immediately revoked. Unlike Redell's capability indirection [22], no special capabilities are seen by the servers and the Java and UNIX APIs can be supported transparently. To revoke capabilities forwarded to other processes (actually their monitors), the monitors must maintain a mapping of capabilities to the processes/monitors that they were forwarded to. Servers must maintain a similar mapping.

Mapping events to permission modifications is done by *transforms* [13, 15]. Transforms map operations to permission transformations. Three types of transforms are defined that implement different methods of transformation (transformation by permission, transformation by membership to an object group, and transformation by combining permission sets). See Jaeger *et al.* for more details [13].

3.3 Process Mediation

Complete process mediation requires that each IPC be authorized by a monitor. We prefer that monitoring be triggered automatically. Otherwise, it may be possible to a programmer to forget to call the monitor (and lose complete mediation). Therefore, there must be a mechanism for redirecting IPCs to the monitor. Then, the monitor must be able determine what permissions are to be authorized. These permissions are authorized by the monitor, and if successful, the requested is forwarded to the destination.

Process mediation requires that the following tasks be accomplished:

- configure the system such that monitors can authorize all controlled operations,

- describe the authorization semantics of operations well enough to enable their authorization,

- provide monitors with the process's current permissions to authorization

Each IPC must be intercepted for complete mediation. Therefore, it is necessary to place a monitor on each IPC path. Typically, monitoring is associated either with the server (which enforces access control on its clients) or on the client (limits access of the client). There are limitations to both approaches. In client monitoring, monitors can, in theory, enforce arbitrary security policy, but in practice they have limited knowledge about the servers to which they are controlling access. On the other hand, servers are typically trusted to enforce security requirements on clients, but they may not understand the security requirements that the monitor is trying to enforce.

We choose a security model that enables both kinds of control, but erlies more heavily on client-side monitoring. Each process may have a monitor that can enforce both its incoming and outgoing IPCs. Therefore, a process that is both a client and a server in different interactions can have its operations to servers authorized and can restrict the operations that it perform for its clients. We do emphasize the client-side control of the monitor by enabling it to place tighter restrictions on the operations it can perform that server monitors may. An additional benefit that results from this model is that monitors themselves can be restricted to different domains because they are different processes. We overcome the problem of server security requirements for processes, by enabling the servers to delegate permissions to content and providing authorization semantics of server operations to the monitors (see below and Section 4.3).

A result of this decision is that an IPC from one controlled task to another requires an additional IPC between the two monitors. However, Lava Nucleus IPC and operation authorization should be fast ($<$ 1.5 μs for small IPCs), so the benefit can be gained at low cost. It is unclear yet whether the cost is worth the added security, however.

The authorization semantics of a server's interface is defined by *operation authorization* objects (see Section 4.3). These objects are used to transform oper-

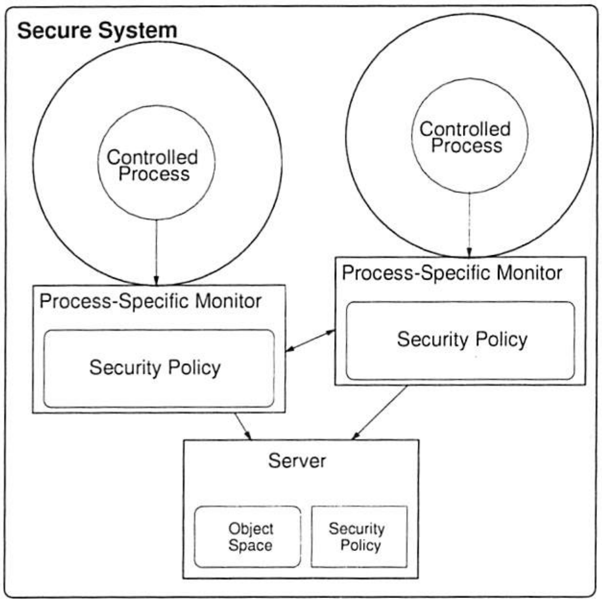


Figure 2: Monitor Architecture

ations into the set of permissions that must be authorized before the operation can be run. These are useful for enforcing least privilege with good performance (we have found the cost of processing these is low).

The monitor obtains authorization operation from the server definition (e.g., via an IDL extension). When a server is loaded, its operation authorization objects are stored in a place accessible to all monitors. The semantics of these operations must be well-understood. Therefore, monitors control client operations (based on the current and maximal permissions) to any server that they are permitted to access.

Our architecture for using monitors and servers to control processes is shown in Figure 2. In this model, a monitor is assigned to each controlled process. When a process makes an IPC, its monitor intercepts the IPC automatically via a kernel-provided mechanism. After the operation authorization semantics have determined the operations that need to be authorized, the monitor uses its process's permissions to authorize the operation. The operation must be within the content's current permissions and maximal permissions to be authorized. Since the content's current permissions may expand and the content's maximal permissions may be restricted, an operation at a certain time may need to be checked against both.

4 Implementation

The security model is implemented upon the Lava Nucleus. The Lava Nucleus provides minimal, general, and efficient functionality for building operating systems. It enables the creation of tasks (i.e., processes) with potentially-overlapping address spaces that may contain multiple threads of execution. An optimized IPC mechanism is provided for intertask communication. The Lava Nucleus also provides a mechanism for IPC redirection which we use to redirect controlled operations to our monitors.

The prototype implementation of this security model is as follows. The Grub boot loader loads the Lava Nucleus and the root Lava task. This task bootstraps the memory system and provides some basic system functionality (e.g., page-fault handling and task-id creation). This task initiates the process load server and the core system services (e.g., a network server to download components and the downloading principal's task). In general, these tasks may also have a monitor assigned to them, but do not at present. The downloading principal's task may request that a new executable task be downloaded by asking the process load server to retrieve the task's content. The process load server has the code authenticated and derives its permissions and transforms. Depending on the load option, the process load server assigns a monitor task to control the new executable. The monitor starts the new executable (or restarts the requestor if loaded into the same address space). Monitors perform both the permission management and authorization services for their tasks using transforms and permissions, respectively.

In this paper, we focus primarily on the implementation of the architecture's monitors. The monitors store the current and maximal permissions of its content, implement its content's permission transformations, intercepts IPCs that are sent by or destined for the its content, determines the authorization requirements of the operation encapsulated in the IPC, and authorizes the operation using the content's permissions. We first describe the monitor's permission representation and how it enables flexible and efficient authorization. Next, we detail the Lava Nucleus's IPC redirection mechanism. Then, we outline how an IPC is converted to the set of operations to be authorized. Lastly, we detail the authorization mechanisms used by the monitor. The monitor uses two mechanisms: a slow one for "binding" to an actual object and a fast one for subsequent calls to the same object.

4.1 Principals and Permissions

Each process in our implementation is associated with a principal data structure. A principal contains references to its complex identity, current and maximal permissions, composition mode, and transforms. The *complex identity* is the composition of principals used to form this principal (e.g., the downloading principal, content provider, application, application role, and session). Current and maximal permissions are as described above. A *composition mode* defines how the permissions of the caller transforms the permissions of this content. For example, a trusted principal may union the current permissions of the caller with their own. However, content not trusted to leak permissions may maintain its current permissions and intersect its maximal permissions. The composition mode is designed to implement permission set transforms (intersection or union of the permissions of two or more principals) efficiently, but other transforms are implemented in a traditional manner because they are not highly performance critical.

To implement both permanent and transient permission set transforms, both the current and maximal permissions consist of static and lexical permissions. Static permissions are the content permissions that apply each time the content is called. Lexical permissions modify the content's permissions based on its current call context. Both enable permissions of other principals to be unioned or intersected with those of this principal (either permanently using the static permissions or temporarily using the lexical permissions). Given that a composition may result in a union, intersection, or no change to the maximal and current permissions either statically or lexically, 24 composition modes are used. Compositions are always performed using the calling principal's static permissions to prevent huge concurrency control overheads that would be likely if lexical permissions were used.

The current permissions are divided into two categories: authorize and active capabilities (maximal permissions are only authorize permissions). Capabilities are software-managed objects stored by the monitors that are unforgeable, revocable, strongly-typed mappings from object identifiers to operations. As we will discuss below, bind operations establish a capability for the content to perform a set of operations on an object. These operations are authorized using *authorize capabilities* that may grant access to more rights than are required for the operation. The result of a bind operation is the generation of an *active capability* that specifies exactly the rights authorized by the bind. Active capabilities enable fast authorization.

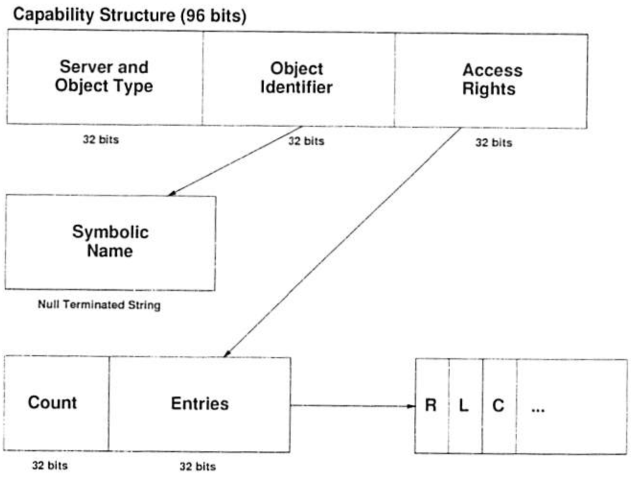


Figure 3: Capability format

Both types of capabilities use the same representation (shown in Figure 3): a server, a type, an object identifier, and the capability's rights. The server and type are captured in one 32-bit field. The server refers to the Lava task identifier for the server (12 bits is the Lava-specific limit). The remaining 20 bits are used to indicate the object type. An *object identifier* specifies the name of the object (reference to a string) or an object reference. References to names are word aligned, so the least significant bit is used to differentiate between the name pointers and OIDs.

The *capability's rights* indicates the operations that the capability grants. Again, a 32-bit field is used. There are 3 status bits, so 29 operations can be granted by default. One status bit signifies that an extended capability rights field is used. Using this field, access to an arbitrary number of rights entries (specified by *count* and *entries* reference) is possible. Each rights entry specifies the operations it applies to (the **R** field) and any limits on the use of these operations (the **L** field with the current count in the **C** field). Another status bit indicates whether the capability indicates a positive or negative access right. The third status bit indicates whether the capability is verified or not. For example, a change in the process's access rights may require a re-authorization of a capability.

Authorize capabilities are unforgeable because monitors will only accept them from process load servers or other monitors (i.e., processes trusted to provide them). They are revocable because the controlled processes never have access to them. Active capabilities derived from them can be revoked, as described below. They are clearly strongly-typed since they have a dedicated type field. However, the servers must enforce this

Active capabilities are formed from the results of the bind operations. For example, the OID is obtained from the object server for fast direct access. The capability's rights are set to those authorized in the bind operation. In addition, active capabilities may have an additional field that stores a unique identifier for the controlled process (called *principal identity*). This field could include a cryptographically secure number that the server can use to identify the calling process in a distributed environment [9].

Active capabilities are unforgeable on a single machine because the monitor is trusted to send them only to the proper server and the kernel is trusted to implement this delivery. In a distributed environment, the capabilities are unforgeable if a cryptographically secure principal identity is sent via a secure channel (i.e., that authenticates the sender). Active capabilities are revocable because they are never given to the controlled process. Instead, they are stored in the monitor, and the only index of the capability (e.g., a descriptor) in the active capability table of the principal is returned to the controlled process. We will see that fast IPC makes this indirection tolerable. Revocation of active capabilities as the result of a revocation of an authorize capability is also possible. Basically, all active capabilities can be marked unverified, so they must be re-authorized before they can be used. Therefore, the monitor must be able to retrieve the original object name (e.g., by caching it).

4.2 IPC Redirection

The Lava nucleus provides a general mechanism that can be used for implementing security policies based on IPC redirection. A monitor can be assigned to multiple processes. Any IPC to a process that is administered by another monitor is automatically redirected to this process's monitor which can inspect and handle the message. This can be used as a basis for the implementation of mandatory access control policies or isolation of suspicious processes. For security reasons, redirection must be enforced by the kernel.

A *clan* is a set of processes (denoted as circles) headed by a monitor. q Inside the clan all messages are transferred freely and the kernel guarantees message integrity. But whenever a message tries to cross a clan's borderline, regardless of whether it is outgoing or incoming, it is redirected to the clan's monitor. The monitor may inspect and/or modify the message. Clans may be nested.

Figure 5 shows a monitor which is used to enforce the security policy. All server requests from the encapsulated tasks are inspected by the monitor (filled circle). The monitor drops any request which would violate the security policy. In particular, it uses accounting mechanisms to restrict denial-of-service attacks. Note that all page-faults and mappings are also handled by IPC. Therefore, the according resources are also under the monitor's control.

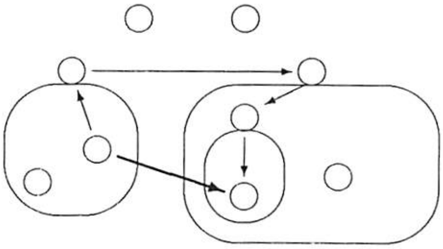


Figure 4: IPC Redirection ("Original" IPC is denoted by thick lines, redirected IPC by thin lines)

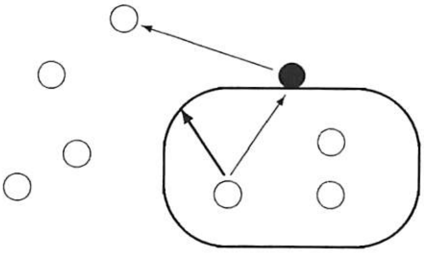


Figure 5: Security-Policy Monitor

Instead of enwalling suspicious subjects, monitors can also be used to protect a system from suspicious subjects outside the own clan. In figure 6 the monitor (filled circle) inspects all messages coming from the outside and drops messages that cannot be authenticated or do not come from trusted partners. Furthermore, the monitor could encipher sensitive messages

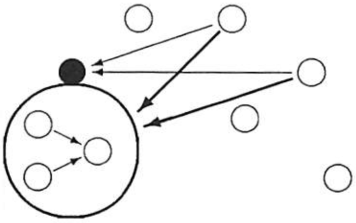


Figure 6: Attack-Blocking Monitor

automatically (i.e., implement secure channels for its clan members).

4.3 Operations

Monitors need to be able to determine how to authorize an operation intercepted in an IPC. Given that new processes may be loaded with new interfaces, the monitors need a standard mechanism for deriving authorization requirements from interface information. This mechanism must be able to determine whether the operation is a bind or active operation, the type of object to which the operation applies, and which operands need to be authorized and the operations for each. The latter is particularly complex for bind operations because the real authorization requirements for the operation are placed in an operand.

We define a object for representing the authorization semantics of an operation below.

- **Definition 1**: An *operation authorization* defines the authorization information for an operation using the following fields:
    - **A Type**: Flag that determines the mechanism used to authorize the operation
    - **Number of Operands**: The number of operands in the operation
    - **Operand requirements vector**: A vector of authorization requirements for each operand consisting of the following fields:
        - **O Type**: Object type for the operation
        - **Ops operand**: Index of the operand that defines the operations to be authorized
        - **Op vector**: A vector of operations to be authorized indexed by the *ops operand* value above or the default value

The actual operation authorization entry used is retrieved from an operations table accessible to all monitors. This table is updated by the process load server (via add only, so no concurrency problems exist). The server and operation number are used to retrieve the entry. The *a type* field indicates the operation type (bind or active) and whether the zero operands, the first operand only, or another authorization is required. This enables fastest path code to be used for authorization. The *operands requirements vector* lists the authorization requirements for each operand. The *o type* specifies the type of the operand. The *ops operand* specifies the operand that determines the operations that are to be authorize. In a file *open*, the second argument determines whether the open is for read, write, and/or append. The *ops operand* may identify that either: (1) the operation requested is to be authorized; (2) a new set of operations are to be authorized; or (3) the operations to be authorized are determined by the value of an operand. For the third case, the *op vector* maps the *ops operand*'s value to the operations to be authorized.

This mechanism should suffice for many UNIX system calls and method invocations. For example, in file *open* and socket *connect* system calls only one operand needs to be authorized. In an object-oriented system, the first operand refers to the only object being operated upon, so only the operations on that object need to be checked. Other objects are passed as OIDs and cannot be accessed unless one of their methods is invoked (and authorized, if necessary).

4.4 Monitors

All IPCs (e.g., page faults, system calls, and RPCs) from the controlled process are automatically redirected to its monitor. The monitor uses its operation authorizations to determine the actual operations to be authorized on the operands. Operations authorizations and principals are stored in two tables shared by all monitors, so they can access any operation's authorization semantics and authorize operations against any principal (which is necessary if a principal's permissions can be lexically modified). Authorization is done using authorize capabilities for bind operations and active capabilities for operations subsequent to a bind.

When a monitor intercepts an IPC, the monitor retrieves the operation authorization to determine the operation's authorization type. Bind operations, such as file system *open*, require that the operation be authorized using authorize capabilities. Any one of the authorize capabilities for that object type may apply, so multiple capabilities may need to be checked. On the other hand, active operations, such as file system *read*, are authorized using the active capability whose index is specified in the operation. Upon a response to a bind operation, an active capability is created which is stored in the principal's active capabilities table. An index to this entry is returned to the controlled process for subsequent use (i.e., a descriptor/OID).

In either case, the operands must be copied into the monitor's address space. Lava supports fast and secure copying of data in IPCs, so such copying can be done automatically [18]. On a 166Mhz Pentium with a 256K L2 cache, up to 8 bytes can be sent in 0.95

μs. 128 and 512 byte messages can be transmitted in as fast as 1.79 μs and 2.60 μs, respectively.

The redirected IPC is then forwarded to the destination where it may be intercepted by that process's monitor. Currently, monitors only authorize outbound operations, but this monitor could restrict the IPCs that can be forwarded from specific sources. For example, if a process's IPCs result in an excessive number of errors (specified by a limit), then the monitor may retract the principal's permissions (e.g., via a transform executed upon an authorization failure).

Once an operation is executed, its results are returned to the controlled process via an IPC (through the monitors). The monitor also intercepts this IPC and may authorize its return. For example, limits on response "operations" can be specified. As yet, we have not exploited this functionality. Currently, the monitor creates active capabilities from the return values of bind operations and returns the active capability descriptor to the caller. For active operations, the return value is simply set in the return value register.

To further improve performance monitors are implemented using special Lava nucleus tasks, called small-address-space tasks. These tasks do not require a TLB flush upon a context switch, so TLB miss costs on a context switch between two small-address-space tasks are reduced (only one TLB miss for the IPC path). We expect that all monitors will be implemented as small-address-space tasks. However, the current implementation of Lava would not support using small-address-space tasks for many small content processes as well, because the number of such tasks is limited to a cumulative address space size of 512 MB in this Lava version.

5 Performance Results

In this section, we measure the performance of the security mechanism as implemented by the monitors. We first make micro-benchmarks of the individual steps in the monitoring mechanism. We then estimate the ideal performance (ignoring effects of TLB and cache misses) of authorized IPC from these benchmarks. We estimate that an authorized IPC (one-way) using active capabilities could be as fast as 4 μs (and typically less than 9 μs ideally). In our current implementation, our fastest one-way IPC is 9.5 μs (authorized using active capabilities). If we have 30,000 average authorized IPCs per second and 10% of these are bind operations (which is a very conservative estimate), then the ideal performance cost is about 20%. We have measured the cost of 30,000 actual active IPCs per second to be 30-40% (9.5 μs per IPC). These numbers are well below the 600% to 800% performance cost for IPC alone for COM objects on Windows NT. Since compiled code may be executed in the remaining 60-70% of the time, the performance impact of IPC interception relative to language-based models may be negligible. In addition, we believe that the number of IPCs can be reduced significantly by judicious code placement (taking security requirements into account) and precreation of active capabilities.

For our performance analysis, we have measured the costs of monitoring operations on a 166 MHz Pentium PC with 256K L2 cache. In our measured scenario, we have two controlled tasks, and each has a monitor that authorizes its IPC. The controlled tasks ping-pong requests back and forth, and we measure the time it takes for the authorized IPCs.

When a controlled process calls an operation the following actions are taken to authorize and forward the operation to its destination task.

1. Prepare the IPC to the destination with the operation data.

2. Send an IPC to the destination that is redirected to its monitor.

3. Determine the authorization requirements for the requested operation.

4. Authorize these requirements for operation and operands.

5. Source's monitor forwards IPC with the operation to the destination that is redirected to the destination's monitor.

6. The destination monitor forwards the IPC to the destination (no control on operation requests is enforced yet).

7. Create active capability descriptor (optional, for responses).

8. The destination receives the IPC.

Operations 1 and 8 are trivial and simply prepare to send an IPC or receive the IPC. Operations 2, 5, and 6 are all basically IPC operations (perhaps with data copying). Operations 3, 4, and 7 implement our authorization mechanism. The costs of all operations except 3 and 4 are fixed for messages of the same size. Therefore, we first list the performance costs of the fixed operations. These values are shown in Table 5. As shown, the fixed costs for monitoring in this configuration vary from 3.03 to 7.98 μs depending on the amount of data to be copied.

Operation	8-byte IPC	12-byte IPC	128-byte IPC	512-byte IPC
1. Prepare IPC	0.12	0.12	0.12	0.12
2. source-monitor IPC	0.95	1.37	1.80	2.60
5. monitor-monitor IPC	0.95	1.37	1.80	2.60
6. monitor-dest IPC	0.95	1.37	1.80	2.60
8. Receive data	0.06	0.06	0.06	0.06
Fixed intercept cost	3.03	4.29	6.58	7.98

Table 1: Performance of fixed interception actions (all times in μs)

A response to an operation goes through the same path, except that a new active capability descriptor may be created for a bind operation (e.g., file *open*). We measured the cost of active capability descriptor creation for a UNIX file descriptor to be 0.69 μs. Cost is kept low by pre-allocating (and reusing) the memory for these descriptors. Of course, active capabilities may be created at content load time to avoid bind operations.

The cost of deriving the authorization requirements is based on the costs of retrieving the operation authorization, determining the operands to authorize, and determining the operations to authorize upon the operands. We compare the costs for evaluating the operation authorizations for two operations: UNIX-style file *open* and file *write*. The *write* operation is an active operation in which only the first operand is authorized for write permission. Therefore, an operation authorization's *ops operand* indicates that the *write* operation is to be authorized and the *a type* indicates that active capabilities are used to authorize only the first operand (0.41 μs). The *open* operation is a bind operation in which the third operand indicates the actual operations that need to be authorized upon the first operand. Therefore, the operation authorization's *ops operand* indicates that the third operand's value determines the operations to authorize (using the *op vector*) and the *a type* indicates a bind operation in which only the first operand is authorized (0.48 μs).

Both the Lava Nucleus and language-based security systems must authorize bind operations (e.g., file *open* and socket *connect*). Language-based systems do not authorize further use of the resultant descriptors (i.e., access operations), so they cannot revoke them. In Table 5, we show the costs of authorizing using both authorization (for a *bind* operation) and access capabilities (for an *access* operation). As will be the case in language-based systems, the cost of verifying operations using authorization capabilities varies based on the size of the object name string (e.g., file path) and the number of authorization capabilities that are examined. In this example, the authorization capability verification does not include additional actions, such as checking inode information. We would expect similar performance for authorization in language-based system given that both systems are optimized.

Scenario	*bind* μs	*access* μs
1 cap-2 byte name	1.70	0.44
10 caps-2 byte name	12.84	0.44
50 caps-2 byte name	56.22	0.44
1 cap-21 byte name	3.50	0.44
10 caps-21 byte name	30.62	0.44
50 caps-21 byte name	138.84	0.44

Table 2: Performance of authorization

Authorization using active capabilities also includes the time for retrieving the active capability from the descriptor (approximately 0.20 μs).

Table 5 summarizes the performance of the Lava security model using address space protection. For access operations, IPC costs range from about 4 to 9 μs depending on the amount of data to be copied. Given 30,000 4 μs IPCs per second, a 12% overhead on processing is incurred. This percentage can be reduced by the percentage of IPCs that can be eliminated by linking content in the same address space. Bind operations can incur a much greater cost (6 to 150+ μs). However, language-based implementations also need to perform the same bind operations (either at load time for a new class or access time for system objects), so the operating system implementation should be faster. In general, since active capability creation is a reasonable cost operation, where possible, it should be used to eliminate unnecessary bind operations.

The actual performance of the entire IPC path for an active operation using 8-byte IPCs is 9.5 μs if the monitors and controlled processes are small-address-space tasks and 14 μs if only the monitors are small-address-space tasks. In the small-address-space case, the additional costs are incurred by the 22 cache misses on data and an slightly higher IPC cost for redirected IPC than the basic IPC benchmarked

Operation (data size)	*Fixed Costs*	*Auth Reqs*	*Auth*	*Cap Create*	*Total*
Active (8 bytes)	3.03	0.41	0.44	0	3.88
Active (128 bytes)	6.58	0.41	0.44	0	7.43
Active (512 bytes)	7.98	0.41	0.44	0	8.83
Short Bind (8 bytes)	3.03	0.48	1.70	0.69	5.90
Medium Bind (128 bytes)	6.58	0.48	30.62	0.69	38.37
Long Bind (512 bytes)	7.98	0.48	138.84	0.69	147.99

Table 3: Performance for different types of authorizations and data sizes (all times in μs)

above. In the large-address-space case, TLB misses become a factor. Despite the performance degradation from ideal, the overall performance for 30,000 IPCs per second is about 30% for small-address-space content processes and 40% for large-address-space content processes.

Unfortunately, we do not yet have performance numbers for handling large data transfers. However, the use of shared memory between monitors and a monitor and its controlled process can reduce the performance impact (if security requirements allow its use). For example, on a *write* operation, only the reference to the data and the data size need to be copied to the monitor for integrity reasons. If the controlled task modifies the data, it has no effect on the security of the operation as long as the operation does not complete before the data is copied to the destination. Therefore, only a single copy of the write data from the destination's monitor to destination is really required. Therefore, 8-byte IPCs can be used on two of the three IPCs in the authorized path. Also, Lava enables flexible memory mapping, so a bind operation that enables a large amount of data transfer may prepare shared memory for implementing such transfers efficiently.

Therefore, we believe that the performance of an operating system-level mechanism for controlling fine-grained is comparable to that of a language-based mechanism, particularly if references are typically passed between processes rather than object data. Since compiled code can execute significantly faster than JIT Java code, it is not unreasonable to estimate that the compiled code can do more processing in the remaining 60-70% of the time than the Java code can do in 100% of the time. Given the additional security benefits of operating system and address space protection (execute compiled code with complete mediation), these security models are worthy of strong consideration.

6 Conclusions

We presented an operating system security model and an analysis of its performance that shows that greater security than that of language-based security models can be achieved with minimal additional overhead for fine-grained programs. This security model enables complete mediation of all content using monitors that automatically intercept IPCs from controlled processes and can enforce security policy upon them. The cost of this interception is perceived to be high, but we have shown that using fast IPC and an efficient authorization mechanism we can perform authorized interception with a reasonable overhead. With little application data available at present, it is hard to estimate the exact overheads, but using micro-benchmarks, we predict an ideal overhead of 12% for 30,000 IPC/s and measure an overhead of 30-40%.

We are not the only researchers working in the area of operating system-level security models for extensible systems. Researchers in the area of extensible kernel architectures have embarked on the development of flexible security services [19, 11]. These systems currently focus on extending server functionality to gain more flexibility in control of client processes. Also, other researchers are focusing on operating system extensions to control downloaded executable content [5, 6]. These system describe how the operating system can enable principals to restrict the rights of their processes. We expect that these researchers will all have to deal with the issues regarding control of multiple fine-grained extensions in the future.

We expect that much interesting research in the future will examine the synergy between operating system and language security models. If a lot of data is to be shared between processes, it is yet to be determined if the best trade-off between security and performance is language-based protection or flexible memory mapping of shared data. Lava's flexible memory mapping enables two processes to share memory in a manner that is still revocable by their monitors. However, language-based protection offers safety (at a cost as well) for data structures within the address space. We predict that the future direction of system and application security will be strongly influenced by the answers to such questions.

Acknowledgements

We'd like to thank Asit Dan, Li Gong, Paul Karger, Ajay Mohindra, and Dan Wallach for their valuable insights. Also, we thank the anonymous referees for their comments which have led to significant improvements in this paper.

References

[1] R. Anand, N. Islam, T. Jaeger, and J. R. Rao. A flexible security model for using Internet content. *IEEE Software*, 1997.

[2] J. P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, James P. Anderson and Co., Fort Washington, PA, USA, 1972.

[3] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the 1995 USENIX Winter Technical Conference*, pages 165–175, 1995.

[4] N. S. Borenstein. Email with a mind of its own: The Safe-Tcl language for enabled mail. In *ULPAA '94*, pages 389–402, 1994.

[5] A. Dan, A. Mohindra, R. Ramaswami, and D. Sitaram. ChakraVyuha: A sandbox operating system for the controlled execution of alien code. Technical Report 20742, IBM T. J. Watson Research Center, 1997.

[6] C. Friberg and A. Held. Support for discretionary role-based access control in acl-oriented operating systems. In *Proceedings of the Second ACM Role-Based Access Control Workshop*, November 1997.

[7] F. S. Gallo. Penguin: Java done right. *The Perl Journal*, 1(2):10–12, 1996.

[8] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th USENIX Security Symposium*, pages 1–14, July 1996.

[9] L. Gong. A secure identity-based capability system. In *IEEE Symposium on Security and Privacy*, pages 56–63, 1989.

[10] L. Gong. New security architectural directions for Java. In *IEEE COMPCON '97*, February 1997.

[11] R. Grimm and B. Bershad. Security for extensible systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 62–66, May 1997.

[12] T. Jaeger, F. Giraud, N. Islam, and J. Liedtke. A role-based access control model for protection domain derivation and management. In *Proceedings of the Second ACM Role-Based Access Control Workshop*, November 1997.

[13] T. Jaeger, N. Islam, R. Anand, A. Prakash, and J. Liedtke. Flexible control of downloaded executable content. Technical Report RC 20886, IBM T. J. Watson Research Center, 1997.

[14] T. Jaeger and A. Prakash. Support for the file system security requirements of computational e-mail systems. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 1–9, 1994.

[15] T. Jaeger, A. Rubin, and A. Prakash. Building systems that flexibly control downloaded executable content. In *Proceedings of the 6th USENIX Security Symposium*, pages 131–148, July 1996.

[16] P. A. Karger. *Improving Security and Performance for Capability Systems.* PhD thesis, University of Cambridge, 1988.

[17] B. Lampson. Protection. *ACM Operating Systems Review*, 8(1):18–24, January 1974.

[18] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 175–187, 1993.

[19] D. Mazieres and M. F. Kaashoek. Secure applications need flexible operating systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 56–61, May 1997.

[20] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A provably secure operating system: The system, its applications, and proofs. Technical Report CSL-116, SRI International, 1990.

[21] J. Ousterhout, J. Levy, and B. Welch. The Safe-Tcl security model, 1997. Available at http://www.sunlabs.com/ ouster/safeTcl.html.

[22] D. D. Redell. *Naming and Protection in Extensible Operating Systems.* PhD thesis, University of California, Berkeley, 1974. Published as Project MAC TR-140, Massachusetts Institute of Technology.

[23] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[24] G. van Rossum. Grail – The browser for the rest of us (draft), 1996. Available at http://monty.cnri.reston.va.us/grail/.

[25] D. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for java. Technical Report 546-97, Princeton University, 1997.

[26] D. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible security architectures for java. In *Proceedings of the 16th Symposium on Operating Systems Principles*, 1997.

[27] E. Wobber, M. Abadi, M. Burrows, and B. Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems*, 12(1):3–32, February 1994.

[28] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

Expanding and Extending the Security Features of Java

Nimisha V. Mehta
The OpenGroup[1]
Cambridge, MA 02139
Karen R. Sollins
MIT Laboratory for Computer Science
Cambridge, MA 02139

Abstract

The popularity of the web has had several significant impacts, two of note here: (1) increasing sophistication of web pages, including more regular use of Java and other mobile code, and (2) decreasing average level of sophistication as the user population becomes more broad-based. Coupling these with the increased security threats posed by importing more and more mobile code has caused an emphasis on the security of executing Java applets. This paper considers two significant enhancements that will provide users with both a richer and more effective security model. The two enhancements are the provision of flexible and configurable security constraints and the ability to confine use of certain storage channels, as defined by Lampson[11], to within those constraints. We are particularly concerned with applets using files as communications channels contrary to desired security constraints. We present the mechanisms, a discussion of the implementation, and a summary of some performance comparisons. It is important to note that the ideas presented here are more generally applicable than only to the particular storage channels discussed or even only to Java.

1 Introduction

The April, 1997 edition of the *Graphics, Visualization and Users Study*[18] reports that the number of web based document authors using Java is increasing. In addition, the authors report that the respondents' belief that they understand and trust Java's security is increasing. This is a self-selecting population, listing their occupations in the fields of **computers, education, management, professional,** and **other**. This last category comprises only about 14% of the respondents, so the great majority of respondents are probably fairly computer literate. One must assume that a much larger percentage of the population as a whole would fall into the **other** category. We must also assume that as the "wired" population increases, it also becomes more naive on average. The reason is that more and more of that "wired" population will come from that part of the population that has less education about computers. In particular, one of the most difficult set of issues to understand and use correctly is those surrounding mobile code and security. Thus, we as technologists find ourselves with a dilemma. On one hand we wish to increase functionality, ease of use, flexibility and apparent simplicity to the user. On the other hand, to do this the supporting mechanisms must become increasingly complex and sophisticated.

Mobile code is not new. We have been moving code in files for years. Interpreted languages such as Lisp and Postscript have been particularly prone to mobility. What is changing is how and when code moves and who has what knowledge of its execution. The Web has made the most significant difference here. The naive user moves from one web page to another, perhaps considering them to be rather static resources. Meanwhile increasing numbers of authors are including Java applets or other forms of mobile code in their pages, to be executed on the page reader's machine. Although the majority of these are not intended to be malicious, some may be, and more may simply be prone to errors, leading to potential dangers for the unknowing user.

In response, Sun Microsystems, the creator of Java, the Java Virtual Machine, the Java Development

[1] This work was performed while Mehta was at the MIT Laboratory for Computer Science.

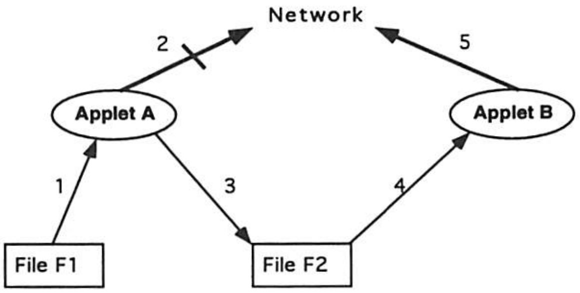


Figure 1: Conditional access and a covert channel: Applet A is prevented from accessing the net if it has accessed file F1. Later Applet B should be prevented from accessing the net after accessing file F2, which was written by A after reading F1.

Kit, etc. has made a concerted effort to close security loopholes. Wallach et al.[23] report on three efforts in the Java environment, but outside Sun Microsystems itself. The other providers of mobile code are addressing the same problems. As a proof of concept we have considered storage channel issues exclusively in Java, although the ideas presented here should be easily portable to other environments.

We began this work with a particular example problem in mind. See Figure 1. Consider applet A that could do useful work for you the reader. In order to do this work it needs to read file F1 and create new files in your home directory, so you would like to be able to give it permission to do that (1). Furthermore, you consider file F1 to be private, so you would like to insure that once A has read F1 it can no longer have access to the network (2). Now suppose that in addition to its job it also creates file F2 (3). Unless you read the code, which is beyond the means of most naive users, you may not know about this additional file. Now at some later time another instantiation of applet A or a second applet B is executed. If the security constraints are not set up properly, this file may be a simple storage channel for exposing some of the contents of F1 to the outside world (4, 5). In the work presented in later sections, we disallow behavior of the sort represented in step 4, in order to close that kind of channel. In Section 6 we discuss how one would allow step 4, but prevent step 5.

There are two significant features to this problem that will be considered in this work: the provision of a system for creating flexible and configurable policies for mobile code, and extending the scope of confinement as described by Lampson[11] to include time-delayed storage channels.

There is an increasing need for a policy language to improve flexibility and configurability of security policies. Wallach et al. describe three schemes for allowing applet access beyond the simple sandbox model. In all three, constraints need to be expressed to describe permissions within the scopes of the schemes. In the capability scheme the propagation of capabilities must be restricted somehow, while in the extended stack introspection approach an access matrix must be generated, and in type hiding there must be a specification of how the type of each object is hidden for each principal. It is worth noting that one can separate the issue of specification of constraints from evaluation of them. In Section 2, other approaches to languages are discussed.

In this work, we have taken the approach in the constraint language of addressing time-delayed storage channels, denial of services, and Trojan horses by means of history-based policies, object-based policies, subject-based policies and floating category labels. The Java Development Kit 1.2 (JDK1.2)[5] provides one such language although not quite adequate for our needs. In parallel with that work, we built ourselves a simple constraint language to meet our immediate needs. Clearly ideas from one can be merged into the other in the long run.

When Lampson described the *confinement problem*, he described three sorts of channels, *storage*, *legitimate*, and *covert.* Moskowitz and Kang[15] address only covert channels and divide them into *storage*, *timing*, and *mixed* channels. What is happening here is that there are two orthogonal axes along which to describe the channels that are at the crux of the confinement problem. These axes have to do with original intention of use and mode of communication. Lampson was interested in distinguishing between channels that were intended to be used as communication channels, albeit perhaps contrary to security policy, and those channels that were not intended to be communication channels. Moskowitz and Kang were moving along the other axis in distinguishing between channels "where the output alphabet consists of different responses all taking the same time to be transmitted" from one in which the alphabet consists of "different time values".

The second problem we are addressing here falls into

Lampson's definition of a *storage* channel, with a twist. The particular problem is that timing delays between input into the storage channel and output may hide the existence of a confinement problem. As a proof of concept, this work focuses on files within this context. We have created two distinct logging facilities one that tracks principals' (applets') accesses and one that tracks applets as file owners. A central component of our approach is the need for a constraint language and set of constraints that are evaluated in order to provide adequate security.

This paper presents concepts. We have designed and implemented them, for demonstration purposes, but the contribution of this work is in the concepts, not the realization. The paper will proceed as follows. Section 2 provides background and presents some of the most closely related work. That is followed by a description of the two logging facilities needed to address the problem with respect to the file system. Section 4 describes our constraint language, followed by a section on implementation issues and some preliminary performance numbers. The ideas in the language could valuably be incorporated into an existing language, such as that described by Gong[5]. It should also be noted that since this effort was generally done as a proof of concept, it was not tuned for performance, although Section 5.5 presents some preliminary performance numbers. The paper concludes with a discussion of further issues related to this work. For a more complete description of this effort see Mehta's thesis[13].

2 Related Work

Any work in the security area is based on an enormous background of previous work. One cannot give complete credit to all the preceding work. Since this work concentrates on extending the security model for Java applets, we will review the security models of the current major Web browsers. We will not address the literature on logging, although there has been a great deal in the areas of systems and databases. We are using only simple, straightforward logging techniques here. In addition, we will briefly examine prior work on authorization languages, followed by comparisons of our work with other secure systems for mobile code. This will include a brief discussion of the applicability of tools that statically analyze mobile code as an extension of authenticating the source of an applet. We will conclude this section with a review of the current situation with respect to Java itself, focusing on the specification of constraints.

2.1 Current Web Browsers

The three current browsers, Internet Explorer 4.0[14] from Microsoft, Netscape's Communicator 4.0[17], and JavaSoft's HotJava[8, 9] provide a variety of mechanisms to authenticate and authorize Java applets. Although none is currently adequate for our needs, they can be extended to satisfy our requirements as specified in this paper.

Netscape Communicator 4.0's capabilities-based feature allows for an extension of an applet's execution space beyond the sandbox. Applets that want permissions beyond the normal sandbox, must notify the browser of the capabilities they require. Unsigned applets are confined within the limits of the sandbox. When a signed applet arrives on the user's system, the user is notified of the identity of the applet's signer and of the capabilities that applet requests. Adhering to the principle of least privilege, the user can then give permission for only that capability. The disadvantage of this design is that the user must give permission for each applet independently. We are concerned that although initially users may consider the permissions seriously, it will not be long before they stop paying attention, and the utility of the authorization will be nullified. Furthermore, since authorizations are given dynamically, this does not allow for the creation of a coherent policy.

On the other hand, Microsoft's Internet Explorer 4.0 (IE) allows users to set permissions statically based on configurable "security zones" from which applets arrive. IE allows the user to set "security levels", permitting extensions beyond the sandbox for certain sets of applets. A **High** level would not allow the applet to go beyond the sandbox, a **Medium** level would allow it but only after warning the user, and a **Low** level would let the applet wreak havoc. For example, applets from the "Restricted sites zone" would be assigned a **High** security level, while those from the "Trusted sites zone" would be assigned to a lower level. Extending the capabilities approach in Communicator 4.0, IE also supports capability signing where the requested capabilities are listed within the applet's digital signature rather than within it's code. Nonetheless, IE is still limited in the configurablity and flexibility of specifying one's policy to preserve confinement and restrict resource consumption.

JavaSoft's HotJava 1.1 supports users in configuring their policies and constraints. As with the other browsers this is based on the signature of an applet, and is then configured by a collection of choices; the user is provided with a checklist of choices. The advantage of this scheme is that the user specifies the constraints only once and is not required to give explicit permission for each execution of each applet. The disadvantage is that the user still cannot specify conditional rules, as is needed to address our problem.

2.2 Authorization Languages

Other research[1, 7, 22] in designing generic authorization languages for secure systems has been done for both military and commercial use, but has not yet been applied to specifying policies for mobile code. This includes work on specifying Separation of Duty policies as done by Sandhu[19], and work on Adage[20] by The Open Group.

Sandhu's efforts on designing control expressions for creating Separation of Duty policies includes a mechanism for maintaining the history of transient and persistent objects using a simple syntax. These policies limit the transactions that can be applied to a particular object based on that object's history. It can be extended to write policies such as: "An applet cannot access a file written by another applet." However, separation of duty policies do not allow one to create a policy which is dependent on multiple objects. For example, it cannot specify our Chinese Wall policy that depends on two distinct objects (networks and files): "an applet cannot connect to the network after reading a protected file."

The Open Group's work on Adage, An Architecture for Distributed Authorization, includes a general-purpose authorization language that is quite flexible and user-friendly. It has an extensive grouping mechanism to categorize subjects, objects, and transactions. It also allows one to create Separation of Duty policies and Chinese Wall policies easily. However, although it maintains a history of transactions, it currently cannot specify policies to limit resource usage. Languages such as these can be applied for creating mobile code policies if they fulfill the requirements specified in this paper.

2.3 Other Secure Systems for Mobile Code

Other projects have attempted to create a secure system with a configurable policy language for mobile code. These include INRIA's SIRAC project[6] on an IDL-based protection scheme for mobile agents, Goldberg et al.'s Janus system[3] for confining untrusted helper applications, and IBM's Aglets[10]. A common goal among all of them is to create a secure system for mobile code that makes use of its features: the collaborative nature of mobile agents or the usefulness of helper applications. However, their common weakness is their limited policy languages.

In SIRAC's scheme for protecting mobile Java agents, the agent's protection policy is defined in an extended Interface Definition Language. This work focuses on protecting access to Java objects between mobile agents by exchanging capabilities between mutually suspicious agents. However, no mention is made of how the language or the system can protect the host's system resources from the agents.

Goldberg et al.'s efforts on confining helper applications involves monitoring and restricting system calls from untrusted applications. Although their approach is language-independent, it is specific to the Solaris operating system. They allow users to specify their permissions ("allow" or "deny") on system calls in a configuration file. However, because the language is quite simplistic, it cannot express policies in terms of resource consumption or histories.

The security model in IBM's Aglets includes an authorization language that allows the agent and its host to specify their policies. Their policy language is rich in that it includes mechanisms to specify the privileges of groups and labelled objects including ways to limit resource usage. However, no further constraints based on the applet's history can be made.

On a different note, others have done research on analyzing the remote code prior to its execution in order to distinguish malicious code from benign programs statically. We have seen this, for example, in the work on *proof-carrying code* (PCC)[16] and on a *malicious code filter* (MCF)[12].

In Necula's paper on PCC the code carries with it an encoding of the fact that it complies with certain invariants or requirements. This constrains the code in meeting various requirements. The sorts of constraints about which Necula is concerned reflect its internal behavior which would be independent of the location at which the code will be executed. In our case, the criteria will potentially be different at each site, making it impossible to provide any proof of meeting useful criteria at its source. If we could offload some of the verification at the source that would be a great benefit, but it is not clear how to do that.

MCF makes use of program slicing and tell-tale signs of system calls to statically test the behavior of mobile code for certain malicious program properties. Unlike PCC, the detecting of tell-tale signs on the client end does not require the programmer to provide a formal specification of the code. One could imagine extending MCF with a policy language to allow a variant degree of code filtering for different applets. Nonetheless, even if these static approaches were extended to be more configurable, they cannot make much use of object-based or history-based policies since such policies can only be evaluated during run-time.

2.4 Security Model in JDK 1.2

Because we are working in the Java context, we must consider the current Java security architecture in the Java Development Kit 1.2 (JDK1.2)[4, 5]. For the purpose of brevity we will not review the basic Java security features of the JDK here, but assume that the reader knows or can learn those easily. We also assume for the purposes of this work, that the Java architecture works correctly, although due to its complexity we realize that the community will continue to find problems that will be addressed by JavaSoft and others.

The goals or objectives were extended in the JDK1.2 to include a simpler policy configuration, a more easily extensible access control structure, and an extension of the security checks to include all levels of Java programs, not just applets. The first of these involves the definition and use of a simple configuration language for statement of policy constraints. The second requires the addition of a **Check** method to the Security Manager in order to support the automatic handling of typed permissions. Finally, the third objective is met by allowing the same sorts of security checking for local code as for mobile code, providing such functions as verification of certification of the code, etc., rather than simply letting local code run completely trusted.

From our perspective we did not want to modify the Security Manager, so having JavaSoft provide the **Check** method would simplify our task. The specification language provides the ability to state only static non-conditional rules. As such its semantics are simpler than ours. There is also no model of past behavior. The syntax is clearly somewhat different. For simplicity of implementation we chose a simple S-expression type language. In JDK1.2, Gong has chosen a syntax that is much more in line with Java's own syntax and allows for the declaration of permission classes. As we will discuss further below, one can easily extend the Java rule specification language with features to support our ideas. We envision a merging of our and JavaSoft's constraint languages into one that embodies the features of both.

Gong highlights in both his papers that one perspective on the model that the new Java security architecture provides is that of security domains. Each is defined by the scope of accesses permitted to a principal. We are doing the same. In carrying that description further, one can describe the problem of storage channels as follows. First, some of those security domains may overlap. These enable the potential storage channels. Second, the security policies in the overlapped regions may be fuzzy, permitting the potential communications channels more invisibility, further enabling the use of them. Our initial approach is to remove the existence of those overlaps. As discussed in Section 6, with a further extension one could allow the overlap, but clarify the policies to be enforced in them and with respect to the non-overlapped sections of those domains. We did not pursue this view further with respect to this piece of work, but it helps to clarify the work on a more architectural level.

The remainder of this paper presents the work that was actually done to address the problem described above. In this context we were addressing two problems simultaneously, that of conditional constraints and that of storage channels, especially those channels which span time by taking advantage of persistent storage in the form of files. In order to do this we found we needed two forms of logging, one to log applets' accesses to files and the other to log applet "ownership" of files. The access log keeps track of activity that is used in the evaluation of conditional rules, while the ownership log tracks files as potential storage channels. The use, realization, and management of these logs is addressed first. We can then describe the language used to express constraints, and finally implementation details and issues, as well as performance. The paper concludes with a discussion of possible extensions to and further thoughts on this work.

3 Logging Facilities

To set a useful policy on applets, users need a way to qualify applets not only by their identities (their origin, their signers, etc), but also by their actions. For example, an applet that only accesses public information can be considered a benign visitor while an applet that tries to modify private system information can be considered suspicious requiring a higher level of security. In order to keep track of such actions by applets, a log needs to be maintained.

Logging will allow users to create conditional policies. For example, when determining whether an applet should be allowed to connect back to its host, a user may want to allow this only if the applet would not be able to compromise the user's privacy. In order to assess this safely, the user would need to determine whether the applet had accessed any private information. (This is better than having our user blindly trust the applet if it came from a trusted source, and distrust it if not.) Providing an auditing mechanism on applets allows users to inspect applets' past histories and check whether any private information was accessed. Of course users are free to allow or disallow all access to their private information, however, this logging feature provides them a way to be selectively restrictive.

Secondly, logging allows a system to keep track of information transfers through storage channels. Such inter-applet and intra-applet communication can be detected by inspecting the past actions of applets. An applet B that reads a file that was written (contaminated) by another applet A can be considered to have communicated with applet A. By this transaction, applet B could have acquired knowledge of any information gathered by applet A. The logging feature allows us to detect and/or prevent such information exchange.

3.1 Logging Accesses

Each time an applet accesses a resource, that action is logged for that applet. An applet's past history can be considered to be the union of all the log entries for that particular applet. To allow the placement of quotas on accesses, each log entry includes a count totalling the number of accesses to a particular resource. For example, one can set a maximum limit on the number of files to which an applet can write, or the number of times network connections are made.

3.2 Logging File Owners

Information exchange between applets via the file system opens the possibility of information leakage from higher privileged applets to lower privileged ones as shown in Figure 1. Such communication can be prevented at transfer 4 by introducing the notion of applet file ownership. This essentially divides the file system into applet domains. In other words, each file written by an applet is associated with its applet owner. Operationally, an applet becomes an owner of a file F once it has written to a pre-existing (but not previously owned) file F, or has created a new file F. This ownership lasts as long as the applet's stored information remains accessible, i.e. until the file is deleted (by the applet itself or by the user.) In the strictest sense, files owned by an applet cannot be read by other applets. This segregation of applets allows us to keep an accurate record of what information an applet has accessed. In our implementation, we have used the conservative approach of preventing communication at transfer 4, however the possibility of allowing file sharing and preventing communication at transfer 5 is discussed in Section 6.

Although information exchange between applets can also occur through other storage channels such as via a network connection, we are more concerned with communication via the file system. We assume that normally users would want to allow applets to continue accessing the file system even after they have accessed protected information, while further access to the network would have been prevented. Hence, keeping a separate log for applet file owners is done for efficiency reasons. However, one can still create a policy to prevent two applets from using a network port as a communication channel.

3.3 Log Storage and Cleaning

As long as the applet is a file owner, an applet's log should last through the stopping and restarting of the applet, and through the exiting and reexecuting of the browser. Once the applet is no longer an owner of any files, it can be safely assumed that it is no longer storing any information from its past history and accesses, and thus it can start from a clean slate. Thus, the algorithm for cleaning entries in the logs is a function of whether the applet currently owns any files.

4 Constraint Language

In order to implement our prototype, we defined a constraint language. Our goals for this were that it address the problem as described earlier of time-delayed storage channels, allow for the control of resource usage, and permit the owner to eliminate certain Trojan horse programs. This was done by providing the ability to write constraints which are a combination of subject-based, object-based, and history-based policy statements. An additional important feature is the ability to assign labels to applets dynamically. Subject-based policies are those based on the identity of the subject or active entity, in this case the applet signature, as embodied in our global applet variables in Section 4.1, as is done in other such languages. Object-based policies are those based on the resource to which the applet wants access. These are also discussed in Section 4.1 in the form of global resource variables. In extending the language significantly, we have added the ability to write constraints in terms of the history of the behavior of the applet, allowing the owner to define denial of service behaviors, and limit them. This feature is realized in the **Any** and **All Past** expressions described in Section 4.2. Finally, our language allows for dynamic labelling of applets based on source or history as described in Section 4.3. The combination of such labelling and histories allows for the identification of certain Trojan Horse applets as well as other untrustworthy applets. The significant strength of this language is the ability to combine these features into constraints.

4.1 Variables

Global variables are provided as a common vocabulary for receiving information and setting permissions. The identity of an applet can be accessed through the variables:

```
Applet.Name,
Applet.CodeBase.Name,
  Applet.CodeBase.Host.Name,
  Applet.CodeBase.Host.IP,
Applet.Document.Name,
  Applet.Document.Host.Name,
  Applet.Document.Host.IP
```

Information about resources can be accessed through the variables:

```
File.Name, File.Path, File.AbsPath,
File.Parent, File.Size, Host.Name,
Command.Name, Property.Name
```

Permissions on resources can be set (true or false) through the variables:

```
File.read, File.write, File.delete,
Host.Connect.To, Host.Connect.From,
Command.Exec, Property.Read,
Property.Write, Window.Create
```

With JDK1.1's *java.security* package, additional variables can be further included to identify applets by their digital signers, etc. In the future, if the JVM can partition the CPU and memory usage by applets, then the variables for **CPU** and **Memory** can also be used.

4.2 Past Primitives

Two boolean procedures are provided for determining the usage of resources in the applet's past history (**Any** and **All**). A procedure was needed that would not only test whether a certain action had occurred in the past, but that would allow the user to request more information about those past accesses. This led to the following syntax for these procedures:

```
(All <identifier> in Past <X> <predicate>)
(Any <identifier> in Past <X> <predicate>)
```

All is used to verify the truth of the *<predicate>* for every *<X>* accessed in the past; while **Any** is used to confirm the truth of the *<predicate>* for at least one *<X>* accessed in the past. *<X>* can be one of two things: 1) a resource or 2) an access. For example, if *<X>* is **File** then it corresponds to all the files that the applet has accessed in the past. However, if *<X>* is **File.Read**, then it corresponds to only the files that the applet has read in the past. The *<identifier>* is used to provide a nomenclature to identify the particular past resource inside the *<predicate>*. When the identifier appears in the *<predicate>*, it assumes the role of the current past resource being tested. Information about the resource can be requested within the *<predicate>* by using any of the variables for resources. For example, if *<X>* is **File**, and the *<identifier>* is **f**, then when **f.name** and **f.parent** appear in the *<predicate>*, they refer to information about the past file currently in question. See the example in Section 4.6 for a sample of this syntax.

4.3 Labels

We have also provided the variable **Applet.Category** for the labelling of applets. This is a modifiable label which can be used to group applets during runtime according to some condition. Having labels makes it easier to identify a set of applets in the access rules. A label can be based simply on the applet's origin such as a label for *is-trusted-to-access-my-mailbox.* More powerfully, a label can be set once an applet has done something in its history. This can be used for keeping track of applets' secrecy levels: *has-read-protected-files*, *has-read-only-public-files*, *has-not-read-any-files.* Also, labels can be applied to applets, as trust in them falls. For example, the label *suspicious* can be set if the applet has accessed more than a certain threshold of protected files. Policies for applets labelled *suspicious* can then be more restrictive. See an example of this in Section 4.6.

Labels have an ordinal ranking which can correspond to secrecy levels. If there are any conflicts in setting an applet's label, the minimum of the label values is conservatively set.

4.4 Primitive Procedures

Various primitive procedures are provided to determine useful information during runtime. These include boolean operators (**And, Or, Not**) and comparison operators (**<, >, =?, !=, <=, >=**). Additionally, procedures that do pattern matching on strings (**Match**) and test whether an element is part of a list (**OneOf**) are provided. Two procedures for totalling the number of accesses to a particular resource are given (**Count** and **CountAll**). For example, **(Count File.Read)** will return the total number of times the applet read the file in question, and **(CountAll File.Read)** will return the total number of times the applet read any file. See the example in Section 4.6 for a sample of their usage.

4.5 Rules Resolution

When a permission is to be checked for a particular resource, the user's applet policy is referenced. For efficiency reasons, instead of verifying all the rules each time an access is to be granted, only the rules that affect that access are verified. The exceptions are those rules that affect **Applet.Category** (they are always re-evaluated) since the applet's label can change at any time. For example, if permission to read a system property is requested, only those rules that assign **Property.Read** and/or **Applet.Category** are evaluated.

In evaluating the access permission for an applet, by default, the access is false. If there are no rules allowing the access, permission is not granted. If there are such rules, the final permission is the "and" of all the permissions set by the rules. So if there is at least one rule that denies access, the permission is denied. This resolves any conflicts that may arise.

4.6 Example

The following is an example of a simple policy that gives applets access to certain directories, to the network, and to the windowing system, while limiting their usage. The number of file writes is limited to 50, the size of these files is limited to 500K, the number of network connections is limited to 20, and the number of windows that can be created is limited to 50. In addition, trusted applets are given read access to some protected directories. The policy also makes use of the **Applet.Category** variable in order to label trusted, contaminated (have read protected files), and suspicious applets.

```
// Define directories.
(Define PublicDirs ("/Public/*"))
(Define ProtectedDirs ("~/Mail/*"
                       "~/Diary/*"))
(Define WriteableDirs ("/tmp"))
(Define ReadableDirs (WriteableDirs
                      PublicDirs))

// Define security labels.
(Define Suspicious 0)
(Define Contaminated 5)
(Define Trusted 10)

// Default permissions on applets.
 // Reading files.
(If (and (!= Applet.Category Suspicious)
         (OneOf File.path ReadableDirs))
    (File.Read = true))

 // Writing files.
(If (and (!= Applet.Category Suspicious)
         (OneOf File.path WriteableDirs))
    (File.Write = true)
    (File.Delete = true))

 // Connecting to the Network.
(If (!= Applet.Category Suspicious)
    (Host.Connect.To = true))

 // Creating Windows.
(Window.Create = true)

// Define trusted applets.
(Define TrustedSources ("web.mit.edu"
                        "lcs.mit.edu"))

(If (OneOf Applet.CodeBase.Host.Name
           TrustedSources)
    (Applet.Category = Trusted))

// Trusted applets get more privileges.
// Allow them to read protected files.
(If (and (=? Applet.Category Trusted)
         (OneOf File.path ProtectedDirs))
    (begin
      (File.read = true)
      (Applet.Category = Contaminated)))

// But keep protected information inside.
(If (=? Applet.Category Contaminated)
    (Host.Connect.To = false))

// Limit number of files created.
(If (>= (CountAll File.Write) 50)
    (begin
      (File.write = false)
      (Applet.Category = Suspicious)))

// Limit file size to 500K.
(If (>= (CountAll File.Size) 500000)
    (begin
      (File.write = false)
      (Applet.Category = Suspicious)))

// Limit connections to the network.
(If (>= (CountAll Host.Connect.To) 20)
    (begin
      (Host.Connect.To = false)
      (Applet.Category = Suspicious)))

// Limit number of windows created.
(If (>= (CountAll Window.Create) 50)
    (begin
      (Window.Create = false)
      (Applet.Category = Suspicious)))
```

5 Implementation

Our prototype that includes the above features is developed on the Sun SPARC platform. We modified the Security Manager of the appletviewer in Sun's JDK1.0.2. Neither the JVM nor the system classes are modified. Our implementation uses the 1.0.2 API of the Security Manager and is built with the 1.1 JVM. In this section, we will first describe the implementation of the Security Manager, the rules, and the logs. We will then highlight a collection of further security issues and conclude with a summary of our performance evaluation.

5.1 Applet Security Manager

The applet Security Manager is questioned by the Java system classes when access to a system resource is requested. When one of the Security Manager's checkX methods is called, it uses the rules and the logs to determine whether the permission should be granted. Not all the checkX methods in JDK1.0.2's appletviewer's security manager were modified. 'checkCreateClassLoader' and 'checkExit' still throw security exceptions for applets since they would otherwise introduce major security hazards. Letting applets create their own classloaders would imbalance the safe foundation set by the lower level security in Java, and allowing applets to halt the JVM seems unnecessary. In addition, checkLink (which unconditionally throws a security exception) is not modified in the current system; however, more flexibility can naturally be provided in the future.

5.2 Rules

The rules are scanned and parsed using Sun's Java Compiler Compiler (JavaCC)[21]. Given lexical and grammatical specifications, JavaCC generates Java code that can parse the rules. The rules must conform to the grammar specified, otherwise a parsing error would be raised. Other errors including mismatched types, global redefinitions, assignments to read-only variables, illegal identifiers, negative applet categories, and so on, are also caught during parsing.

5.3 Logs

The applet file owner logs and the applet access logs are implemented using cached hashtables. The sizes of the two caches are specified as constants which can be easily modified. For now, they are arbitrarily set to size 16. The caches use a least recently used replacement policy.

In order to account for failure in the system, all the data in the cache is written back to the log after a certain number of events or minutes. In case the system fails or is exited abnormally, these regular writebacks will prevent major loss of information. The loggers include constants to specify the maximum number of events and the maximum number of minutes between writebacks. If the application exits normally, writebacks are also done upon exit.

5.4 Security Notes

In the implementation of the system, certain security issues needed to be addressed. As in Java, a safe design is only a support structure for a secure implementation. Five are highlighted here.

5.4.1 Error Resolution

How does one resolve various errors? Errors include I/O errors, parser errors, applet security errors, other runtime errors (i.e. out of bounds, unknown host), and other unexpected system errors. Since this system involves the maintenance of multiple simultaneous running applets, we need to classify these errors into *fatal system errors* and *applet-specific errors.* The first class of *fatal system errors* includes those errors where the entire system is halted, since otherwise, security violations would occur. If the system is inside a web browser, then the browser should halt all its Java operations when encountering a *fatal system error.* The latter class of *applet-specific errors* includes those errors where only a particular applet's execution is halted. These errors that affect only one applet should not halt the entire system. If the entire system were halted from such an error, then that would allow an applet to affect the execution of other applets by simply causing those errors (denial of service attack). It seems apparent that only those errors which affect all applets and the security of the entire system should be considered *fatal system errors.* This includes parser errors and I/O errors from reading the rules, and format and I/O errors from the file owner log. On the other hand, a security violation by a single applet and a formatting or I/O error in a single applet's access log need not affect the entire system. Instead, these errors are signalled for the benefit of the user and the execution of that applet is terminated.

5.4.2 Applet Identification

How does one identify applets? In the appletviewer, the host of the applet's document is most commonly specified using its DNS host name. In so doing, our implementation simply identifies an applet by the host name of its codebase. Since a server sometimes uses multiple machines (and thus perhaps multiple IP addresses) to reduce its load, our prototype does not distinguish the same applet on the different machines. Identifying an applet by its host name (and not its IP address) allows the applet to later access its files. However, this leaves room for DNS spoofing attacks, in which incorrect entries in a DNS server lead to incorrect identification of applets.[2] If the DNS server becomes infiltrated since the last execution, then the correct identity of an applet changes. This introduces a vulnerability to an external source: the DNS server.

On the other hand, the support for digitally signing applets can address this. Instead of IP addresses or DNS host names, the signature on the applets would provide a secure mechanism for identification. This functionality has now been included in JDK1.1's *java.security* package. However, one may want to consider an applet that is updated to a newer version to have the same identity as its older version so that it may access its old files. This would require a slight variant to digital signing where the identity of the applet does not change with slight modification to its code if the author and the origin remain the same. Although this problem needs to be addressed, we do not address it further in this work.

5.4.3 CheckX methods

The checkX methods in the Applet Security Manager are called when system classes want to verify whether the current thread (applet) has the authority to access a certain resource. These methods are provided to check access. However, calling a checkX method does not necessarily mean that the given applet has performed that action, but only that the action was requested. Despite this, the current implementation logs it as if the action was performed. For example, the java.io.File.canRead method uses the checkRead method to just check whether the read permission should be allowed, although the file may not even be read.

Web browsers can also give applets access to the Security Manager, enabling applets to inquire about their permissions using the Security Manager's checkX methods. This feature allows applet authors to write more robust and useful code. However, with our current implementation, the checkX methods will log these inquiries as actual accesses. The outcome is that applet's accesses would be limited by these extra loggings if the applet policy includes rules that limit future accesses based on past ones. For example, if a rule states that "an applet cannot access more than 8 files," the applet is able to access only 4 files since the author's checks prior to each access would also be counted. This does not introduce any security holes if past accesses only limit future accesses. If this were not the case, then an applet could merely call the checkX methods without actually accessing the resource but instead extending its permissions. An example would be a rule that states, "an applet can access this protected file only if it has read this copyright." The applet could simply call the checkRead method on the copyright, but not read it.

To address this properly however, the Security Manager in the JDK should have two types of methods: one for checking access (checkX) and another for both checking and actually making the access (checkLogX). The former can be used by applets that want to know their permissions, and the latter should be private to the system classes.

5.4.4 Eliminating Race Conditions

One needs to make sure that the system is not faced with the same flaw as the one in *fingerd* where a malicious attacker exploits the race condition between checking the properties of an object and giving the permission. Applying this to our scheme, let us say there is a rule that states that an applet cannot connect to the network if it has read a local file. Then between the time the SM checks whether a thread can access the network and the time when it gives the permission, another thread of that applet reads a file. So in the end, the multi-threaded applet was able to circumvent the rule. In order to solve this problem, one must synchronize the Security Manager's check for accesses by applet rather than by thread.

5.4.5 Unix File System

We have currently limited our focus to the UNIX file system. To be consistent with the capabilities on the UNIX platform, giving write access to an applet does not mean it has read access. In our implementation, in order to allow an applet to read and write, both permissions must be assigned to true in the rule.

On another note, the JDK API is limited in that the file permissions on the UNIX system are inaccessible. Therefore, there is no way (other than writing native code) of setting a file's ACLs or discovering whether a file's SUID bit is set. This limits the breadth of the policy one can place on an applet's access to the file system. Further, any files that are newly created by the Java runtime are given ACL permissions based on the user's current *umask* value. If the *umask* value does not prevent the creation of world-readable files, then all new files would be world-readable. The file permissions cannot be changed after its creation because of the limitation in the Java API. Consequently, the log files created by the system and files created by applets are dependent upon the *umask* value.

5.5 Performance Analysis

We have analyzed the performance of our implementation by executing an applet that reads a line from a file and writes a line to a file a certain number of times. We measured the amounts of time for reading and writing 1250 times, 2500 times, and 5000 times on a Sun Sparc 5. The experiment used a sample policy that gives trusted applets access to certain public directories while restricting the file size to 100K and the file writes to 50. This simple test involved logging the past accesses, checking the past accesses each time a read or write was to be done, and analyzing the rules during runtime. We compared our times with the times for JDK1.1's appletviewer whose Security Manager needed to be slightly modified in order to allow reads and writes to the file system. The results show that the amount of time our extended system takes is 1.67 times that of the regular appletviewer. In particular, our prototype takes .131 seconds for each additional iteration compared to the regular appletviewer's .079 seconds. Performance can be improved with further work in providing additional JVM native support and by using a Just-in-Time compiler.

6 Conclusion

This paper has addressed the issues that arise with making applets less restrictive by giving them more access to a user's operating system. We have attacked this problem by 1) supplying a constraint language that can specify conditional rules based on past actions and 2) monitoring the actions of applets through logging facilities. With these two features the information exchange described in Figure 1 can be easily detected and prevented.

Our implementation isolates applets by the notion of file ownership and by disallowing applets from reading files owned by other applets. However, as a future extension of our work, this restriction can be lifted if the sharing of files among applets is needed. Such a capability would be useful if one wants to implement applets that collaborate. For example, as one applet organizes and arranges a user's schedule, another could graphically present the scheduler, while another could communicate with other agents to make appointments. These teamplayer applets would need to communicate with each other and would need to share the common schedule files. With file sharing in place, the communication control would be pushed down to step 5 of Figure 1.

One possible secure implementation of file sharing would require associating a static security label with each file in addition to the dynamic label associated with each applet. The label on the file would denote the security level of its contents, while the label on the applet would correspond to the highest security level of the information that it had accessed up to that point. Then applets with the same security level could access the same files without compromising the local system. This way, if applet communication occurs through the shared files, the applets would have accrued the same security level of information. Such an implementation would require extending the constraint language to allow users to specify the security levels of files in a straightforward way, so that the rules would be less prone to error.

Although in this paper we have primarily addressed issues with applets communicating via the file system, there are also other storage channels through which applets can communicate. These include the method calling between applets from the same document through the procedures *getAppletContext* and *getApplet*, and the spawning of new applets on the local file system. More details about these storage channels and how our prototype addresses them can be found in Mehta's thesis [13].

In conclusion, we believe that the addition of conditional rules referencing past actions and complementary logging facilities will add significantly to the usability of the Java security mechanism. These features will also allow us to address the storage channels that exist in the system. Furthermore, these features can be easily portable to JDK1.2 and other mobile code systems. We have demonstrated reasonable performance of this functionality in a prototype implementation.

7 Acknowledgments

This work was supported by the Department of Defense Advanced Projects Research Agency under contract number DABT63-94-C-0073 for work done at MIT's Laboratory for Computer Science.

References

[1] D. D. Clark, D. R. Wilson, *A Comparison of Commercial and Military Computer Security Policies*, **IEEE Symposium on Security and Privacy**, Oakland, CA, April 1987, pp. 184–194.

[2] D. Dean, E. W. Felten, D. S. Wallach, *Java Security: From HotJava to Netscape and Beyond*, **IEEE Symposium on Security and Privacy**, Oakland, CA, May 1996, pp. 190–200.

[3] I. Goldberg, et al. , *A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker*, **USENIX Security Symposium**, San Jose, CA, July 1996, pp 1–13.

[4] L. Gong, *Java Security: Present and Near Future*, **IEEE Micro paper**, **17**(3), May/June 1997, pp. 14–19.

[5] L. Gong, **Java Security Architecture (JDK1.2)**, Rev. 0.5, July 10, 1997.

[6] D. Hagimont, L. Ismail, *A Protection Scheme for Mobile Agents on Java*, **ACM/IEEE International Conference on Mobile Computing and Networking**, Budapest, Hungary, 1997.

[7] S. Jajodia, *A Logical Language for Expressing Authorizations*, **IEEE Symposium on Security and Privacy**, Oakland, CA, May 1997, pp. 31–42.

[8] JavaSoft, Sun Microsystems, *HotJava(tm): The Security Story*, May 1995, http://www.javasoft.com:/sfaq/may95/ security.html.

[9] JavaSoft, Sun Microsystems, *HotJava(tm) Browser, Version 1.1 Beta2* http://www.javasoft.com:/products/hotjava/ 1.1.

[10] G. Karjoth et al, *A Security Model for Aglets*, **IEEE Internet Computing**, **1**(4), July/August 1997.

[11] B. Lampson, *A Note on the Confinement Problem*, **Communications of the ACM**, **16**(10), October 1973, pp. 613–615.

[12] R. Lo, K. Levitt, R. Olsson, *MCF: A Malicious Code Filter*, **Computers & Security 14** (6), 1995, pp. 541–566.

[13] N. V. Mehta, **Fine-Grained Control of Java Applets Using a Simple Constraint Language**, MIT/LCS/TR-713, June 1997. Also thesis for Master's of Engineering, MIT. June 1997.

[14] Microsoft Corporation, *Microsoft Security Management Architecture White Paper*, May, 1997, http://www.microsoft.com/ie/security/ie4security.htm.

[15] I. S. Moskowitz and M. H. Kang, *Covert Channels - Here to Stay?*, **COMPASS '94**, Gaithersburg, MD, June 1994, IEEE Press, pp. 235–243.

[16] G. Necula, *Proof-Carrying Code*, **ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages** Paris, France, January 1997, pp. 106–119.

[17] Netscape Communications Corporation, *Securing Communications on the Intranet and Over the Internet*, July 1996, http://www.netscape.com/newsref/128bit.html.

[18] J. Pitkow and C. Kehoe, **GVU's 7th WWW User Survey**, Georgia Institute of Technology, April 1997, http://www.cc.gatech.edu/gvu/usr_surveys/survey-1997-04.

[19] R. Sandhu, *Transaction Control Expressions for Separation of Duties*, **4th Aerospace Computer Security Conference**, December 1988, pp. 282–286.

[20] R. T. Simon, M. E. Zurko, *Separation of Duty in Role-Based Environments*, **Computer Security Foundations Workshop**, Rockport, MA, June 1997.

[21] Sun Microsystems, Inc. , Java Compiler Compiler, Version0.6(Beta), 1997, http://www.suntext.com/JavaCC/.

[22] L. van Doorn, et al., *Secure Network Objects*, **IEEE Symposium on Security and Privacy**, Oakland, CA, May 1996, pp. 211–221.

[23] D. S. Wallach et al., *Extensible Security Architectures for Java*, **Symposium on Operating Systems Principles**, St. Malo, France, October 1997.

Towards Web Security Using PLASMA

Annette Krannig
Fraunhofer–Institute for Computer Graphics IGD, Germany

Abstract

The World Wide Web is one of the most significant multimedia applications ever developed — and therefore securing the web is one of the most pressing problems. There exist a number of approaches for securing the World Wide Web which, however, usually pursue what one might call a *low level approach* without being able to give adequate consideration to the specific requirements of this multimedia (or hypertext) system.

The subject of this paper is the realization of an adequate security system, which is capable of detecting the different media and structures within hypertext systems and therefore apply different cryptographic mechanisms to them; this resulted in the development of the system PLASMA (**Plat**form for **S**ecure **M**ultimedia **A**pplications).

PLASMA is a security platform designed within the frame of the Berkom R&D-programme at the Fraunhofer–IGD in Darmstadt whose prototype was developed to provide a means for secure multimedia telecommunications. In order to demonstrate the capabilities of PLASMA, it was integrated into a W3 scenario. The advantages of PLASMA when used in the World Wide Web as well as the architecture created for the integration process are described in the following section.

Keywords

World Wide Web, High level security, Multimedia, Secure communications platform

1 Introduction

Secure telecommunications is a subject which has been addressed extensively in the past; the same is true for the secure World Wide Web. An overview of this topic may be found in the paper by P. Lipp and V. Hassler [10]. Protocol realizations such as SHTTP [14] or SSLeay [13], an implementation of SSL [5], are examples of the prior art in this field.

Yet even in 1996 B. Fernandez noted in [3] that these works concentrated primarily on *low level security*; these approaches do not address the multimedia or structural elements of the application documents. This is an area which has received scant attention in the past and on which this paper tries to shed some light.

From [3]: "Hypertext systems encompass multimedia documents with hyperlinks connecting to other similar documents. Multimedia documents and their hyperlinks may contain a variety of media including audio, video, animation, graphics and text. Extensive research is going on about hypertext/multimedia systems as they are the upcoming platform for worldwide transfer of information. With this rapid growth in hypertext documents and WWW sites, their security is a top priority issue. Compared with other topics, this is a relatively neglected aspect. This is strange because the richness and variety of multimedia information provides with many opportunities to access unauthorized information. Most of the

> work until now has focused on methods to protect the information or authenticate the users using cryptographic measures. While these are certainly useful, they are too low level to control access based on logical properties of the document information."

To overcome these deficiencies a security platform PLASMA for multimedia telecommunications and applications has been designed and implemented for differentiating between media and structural components and using different cryptographic algorithms accordingly. For the purpose of demonstrating the functionality of PLASMA, the World Wide Web was chosen. The reasons for this were that on one hand more and more transactions occur via the web, making the WWW one of the principal applications in electronic communications overall; on the other hand the World Wide Web constitutes a multimedia system as well as a hypertext system with components which can be distinguished by the structure of the application documents.

PLASMA was a project within the R&D–programme of Deutsche Telekom Berkom GmbH, which is a subsidiary of Deutsche Telekom AG. Deutsche Telekom has developed a product version of PLASMA which is based on Deutsche Telekom's security technology as well as the results of the PLASMA research project. PLASMA is currently available for several operating systems. In this contribution all mentioning of PLASMA refer to the PLASMA research project and the PLASMA prototype realized in this project.

2 New requirements for a secure WWW

The World Wide Web is a multimedia hypertext system; it consists of text documents including images as well as a host of other documents of varying structure and differing media types.

For example, single words or simple buttons in a hypertext document may refer to different documents, images or Java applications which are activated by a mouse click. If the so–called *hyperlink* is activated, the corresponding web server is contacted and the selected *document* is requested by the client. This *document* may be another text document, an image, a Java applet simulating a video or a so–called *form*. *Forms* are sent to the client by the server, filled out by the client and then returned to the server.

In the World Wide Web, text, images, Java applets, forms etc. are transmitted, suggesting the use of different cryptographic mechanisms for these different media types and structural elements. Image and video data may be protected differently from textual data. For example, it is often sufficient to simply reduce the image quality of the image data or to encrypt merely a segment of the image (e.g. encrypting a face). Furthermore, image and video data are of considerably larger volume than textual data which may make the use of faster cryptographic algorithms (which will then be usually less secure) necessary. The divergent media and structures of an application document or the various document types should also be treated differently by a security platform.

Another layer of security is the protection of the entire document during transmission. A document of a multimedia application is usually a composite of different media types; yet it must be considered as a composite whole.

It is necessary to specifically associate media and cryptographic protocols which determine the appropriate algorithm for each media type. However, once the appropriate cryptographic mechanisms for a media type within a document have been determined, they cannot be changed within that document context.

The selection of these security services should be performed by the user since only he can decide on the basis of a specific document whether it should be signed or merely be encrypted in transit — this makes user interactions necessary.

Both are problems which are hardly if at all considered in the current telecommunications systems yet are realized in PLASMA. As a demonstration of these features, PLASMA was integrated into the World Wide Web. The thus developed solution will now be described. The architecture to be presented itself does not offer new insights into the subject of web security but does demonstrate a meaningful application for the security platform PLASMA in the World Wide Web and the new concepts realized within the platform, namely the idea of *high level security*.

Furthermore, this implementation may be used in a real–world scenario for securing communications in the World Wide Web since

- PLASMA allows securing several simultaneous communication sessions; this allows a web server to serve several clients simultaneously using secure connections; similarly a client is capable of starting several simultaneous requests to different servers.

- It is possible to create the presented scenarios (and the architecture presented therein) even when firewalls (of the packet filtering/stateful inspection variety) are used to secure a corporate intranet. PLASMA uses the standard HTTP protocol for tunneling its secure requests and replies thus facilitating real–world secured web communications[4].

- A basic functionality of any kind of cryptographic systems is a correct key management, i.e. the proper administration and storage of the cryptographic keys for each user. The current realization of the key management subsystem in PLASMA conforms to the X.509 authentication framework [15], thus making for a clean implementation of this vital part of a cryptographic system.

3 What is PLASMA?

3.1 Media and structure specific cryptographic operations on application documents

PLASMA is a security platform which is capable of differentiating cryptographically between the various media — the underlying engine also allows different structural components of a document or different document types as they may occur in hypertext documents in the World Wide Web to be operated on using different cryptographic mechanisms.

What are the requirements for this – The *Filter* module The paramount requirement for the above mentioned procedure is that the application (i.e. the multimedia or hypertext system) is capable of handing down information on media structure, structural elements or document types to the security platform. An application document therefore must be split into its multimedia or structural components which in turn must be passed on in PLASMA–specific formats to the security platform. This task is performed by the *Filter*.

The *Filter* is a module that is required by the security platform yet logically resides within the application since only the application has the knowledge required on format structures which vary from application to application and may even diverge strongly from prior versions of the same application. Therefore each application must provide its own PLASMA *Filter*. The data structures within PLASMA for multimedia documents are constant and can be addressed by each application in a similar fashion; it is the application formats which cannot be addressed adequately in a general form by PLASMA.

A *Filter* dissects a multimedia document of the application into its media specific components by noting the locations of subcomponents (e.g. the locations of textual and image parts) into a *Script*; the media specific parts of an application document are thus transformed into their corresponding media specific data objects within PLASMA[1].

If after reverting the cryptographic operations these media specific data objects of PLASMA are to be transformed back into the plaintext document, the *Filter* requires this *Script*. The *Script* provides the required information for the *Filter* to assemble the plaintext document into its original form; the PLASMA specific text objects are written into positions originally containing text portions; similar procedures are followed for image data and other media specific objects within PLASMA. Figure 1 illustrates the concept of a *Filter*.

[1] The *Script* is stored together with the other media specific data objects into a *Container* by PLASMA — with the *Container* being a collection object maintaining its subobjects as a list.

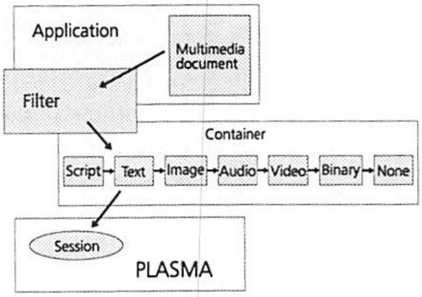


Figure 1: The functionality of a *Filter*

PLASMA thus obtains media specific data objects from the *Filter* which are then to be processed using various cryptographic mechanisms.

What further modules are required? PLASMA provides three security services for data transmissions in its current implementation; they are *non-repudiation*, *confidentiality* and *integrity*. The object–oriented design of the security platform allows the easy integration of further security services such as the *protection from replay attacks* or the *digital signature creation*[2].

For PLASMA to be able to perform media specific operations on the application document data it is necessary that the security platform can call upon different and media specific protocols for the actual operation; this is realized by the concept of *generic security services*. A generic service *non-repudiation* not only provides one fixed protocol for realizing a non-repudiable component but rather has access to a number of different protocols which then perform the actual operation[3].

The security platform must be capable of activating the particular security services for each media type or structural element and their corresponding cryptographic protocol — these can and should be media specific protocols.

The simplest and most obvious approach for this is to implement the corresponding parameters as self–sufficient modules of the platform. This results in the existence of the modules *medium*, *generic security services* and *cryptographic protocols*.

This architecture necessitates another module which creates the relations between these parameters; this relation must express which cryptographic protocol is to be used for a specific medium, structural element or document type with regard to an activated security service. This relation is designated within PLASMA as the *security policy* since this relation actually describes the policy or mode of operation how different media are to be protected cryptographically during transmission — it must in some form be capable of influencing the relations between the other three modules *medium*, *generic security services* and *cryptographic protocols*.

The security policy in PLASMA is stored for each user as an ASCII file in the home directory of each user (*FSP: formal security policy*), thus allowing each user to configure this policy to meet his needs prior to the secured communication itself. The actual use of these security policies in an ongoing communications session, their enforcement and application will be explained later.

The rough outline of the security platform is therefore predetermined to appear similar to Figure 2 at the left side. It is obvious that further modules are required for the implementation of "secure telecommunications" when considering the entirety

[2] The services *non-repudiation* and *digital signature creation* do not differ from a purely technical viewpoint; however, in the latter case the user must be given an opportunity to actively confirm that he wants to sign the given document.

[3] A cryptographic algorithm is referred to herein as a protocol since for example in the case of the DES algorithm the protocol for reverting the encryption on behalf of the recipient is well defined.

of a security platform; such objects as the *Session* object depicted in Figure 1 which implements a secure connection between two communications partners within PLASMA — however, a detailed description of the object oriented design of PLASMA was already given in [7] and shall not be repeated.

How are the modules activated? The *generic security services* module could be activated by user activity or alternatively, they might get activated sequentially. The module *cryptographic protocols* must interact with the underlying security technology which provides the actual implementation of the cryptographic algorithms. The *medium* module obtains its data from the *Filter* and differentiate the data stream into several classes. The *security policy* can be set directly by the user or by a security administrator.

3.2 Secure telecommunications using PLASMA

What are the basic requirements for secure telecommunications which are, of course, also realized in PLASMA? First of all each user trying to establish a secure connection to one or multiple communications partners requires access to his personal security–relevant data. These data are stored in a specially secured area, access to which is possible only using a valid PIN.

PLASMA requests this PIN upon establishment of a secure connection from the user. If the PIN entered is correct, the storage area (which can best be thought of as a SmartCard) is opened and the user is then enabled to create a secure communications session; otherwise the establishment process is stopped at this point.

After this access control for the personal security relevant data of the user which in particular contain cryptographic keys it is also necessary for the user to gain certainty about the identity of the communications partner; this requires a mutual authentication of the participating parties in the best case, for which several protocols exist. In PLASMA, this is implemented by means of the X.509 authentication protocol [15].

After a successful X.509 three-way-authentication all participants are well informed about the identity of their counterparts: The web server knows which client wants to access his data and conversely the client knows he has contacted the correct web server — such information is vital, particularly in the case of online transactions and electronic commerce.

During the authentication the security policies, the public asymmetric keys including certificates as well as the session keys which are later used for transferring confidential data are exchanged[4]. The concept of security policies, the exchange thereof in the authentication phase and the subsequent reconciliation of the policies of both communications partners are significant elements of PLASMA: an application document is processed by PLASMA according to the rules extracted from a *Coordinated Security Policy CSP*. The *CSP* is calculated from the security policies (*FSPs*) of the communications partners which are exchanged during the authentication phase.

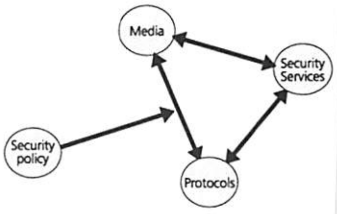
A security policy which yields a relation between the modules medium/structure, cryptographic protocols and security services, thus assigning a cryptographic protocol to a combination of a medium and an activated security service is partitioned into two segments: the *profile* and the *rule* part.

The profile part lists the cryptographic protocols available for each security service; this is useful since it may be the case that one of the communicating partners is willing only to communicate if SmartCards are used for storing the cryptographic keys or even that one side wants to exclude a simplistic cryptographic algorithm from the negotiation since it is deemed too insecure by that party.

The rule part lists the specific cryptographic protocols which are to be used when encountering a given media or structural type of an application document in order to provide a requested security service[5] (cf. Figure 2, right side).

[4] For the necessary background of cryptographic material refer for example to [15].

[5] The "protocol" *None* is for document parts which should not be treated cryptographically.



Media	Non-Repudiation	Confidentiality	Integrity
Text	*RSAwithMD5*		*MD5*
Image		*DES*	
Audio			*None* *MD5*
Video		*None*	

Figure 2: Relevant modules and a possible security policy

These security policies are created locally at each user's site and must be reconciled prior to the communication proper and immediately after the mutual authentication with communications partner; result is the *CSP*, mentioned before. To achieve this the set intersection of the profile parts of both policies is generated; in the case of the rule parts the set union is created. This way only such protocols are allowed for the communication which are available on both sides of the communications channel and only those cryptographic protocols are selected which do not contradict any rule from either party.

After a successful authentication of the communications partners the secure communication itself between the participating parties may commence. To achieve secure communications, the security services confidentiality, non-repudiation and integrity are made available by PLASMA[6].

Using PLASMA it is now possible to send messages with guaranteed integrity, i.e. the message will arrive provably at the recipient as it has been sent by the sender; non-repudiable message passing means that the sender of the message is uniquely identifiable by the recipient – beyond that the non-repudiation automatically includes the integrity of the transmitted document; lastly, PLASMA is capable of generating confidential messages, i.e. messages which only authorized parties – the sender and the recipient – are able to decrypt; all other parties can only access the message in encrypted form.

The concept of security polices and, correspondingly, the media–specific operations which also include the structural properties of an application document are now brought to the fore in the following: the generic security services are activated by default sequentially by PLASMA or after explicit user interaction — only the user may decide at runtime whether a document to be transmitted shall be signed or protected against a loss of confidentiality.

Consequently, a media specific cryptographic protocol matching the activated security service and current media type must be selected which will then be used to meet the security policies' goal. To achieve this, the security policy *CSP* is queried; this object is structured as a table assigning each media type and generic security service a specific cryptograpic protocol (cf. Figure 2 at the right side).

The result of this query is a cryptographic protocol which is subsequently activated by PLASMA. The data are transmitted to this protocol which will then perform the cryptographic operations specific to itself on these data.

Lastly, an integral part of each secure communications session is the ability to dismantle such a connection properly in such a way as to disallow a possible intruder the disruption of such a connection. Therefore this functionality has been integrated into PLASMA.

The possibility of user interaction is, similar to the media specific operation on an application document, only possible if the security platform is situated logically close to the application and is capable of communicating with the application. This goal has been reached within PLASMA by taking the approach of locating the platform close to the application based on the concept of the GSS-API [9], [17].

[6] Through the object oriented design of the security platform PLASMA, as described in [7], it is easily possible to integrate further generic security services into the platform.

An essential feature of any kind of security (in the cryptographical sense) is the need for key management, i.e. the proper administration and storage of the cryptographic keys for each user. PLASMA is based on a security technology which implements the cryptograpic algorithms (cf. [7]). The implementation used as a security technology in PLASMA is SecuDe (cf. [16]); it is also possible to implement it using a different security technology, for example utilizing the widely spread security package PGP [18].

Therefore the key management of PLASMA is fundamentally dependent on the functionality of the underlying security technology. If these security technologies offer key management themselves, PLASMA can merely integrate it into itself; otherwise it needs to be modeled within the platform itself; the security technologies PGP and SecuDe for example offer proprietary utilities for certification of their keys and also define the makeup of certified keys and certificates[7].

4 The integration of PLASMA into the World Wide Web

There are two well–distinguished layers in this W3 scenario. The first layer is that of the World Wide Web and is situated between the browser, the proxy and the CGI programs. In this layer, the client poses requests to the server by activating a hyperlink; this hyperlink contains a call to a CGI program which creates the server's response: a HTML page which in turn contains a hyperlink to the next CGI program in the sequence; this is the layer of the HTTP protocol.

The second layer is located between the PLASMA applications on the client and server sides. Here, data are passed through to the security platform so that PLASMA may perform cryptographic operations on them. The protocol to be used between the PLASMA applications is predetermined by the application independent API of PLASMA and may be considered separable into three phases: The connection establishment and authentication phase, the secure document transmission phase, and the connection teardown phase.

When integrating PLASMA into the WWW the principal problem is to pass the data on both sides from the WWW layer onto the PLASMA layer. On the serverside this is possible using CGI programs (Common Gateway Interface [12]). When using a Mosaic browser, a CCI implementation (Common Client Interface [11]) on the client side and CGI programming on the server side would be possible; but here it is not possible to encrypt a client's request[8]: The client formulates his request by activating a hyperlink to a server side page. This causes the service request to be forwareded immediately to the server. In *addition* to that, the request can also be transmitted to the CCI interface "in parallel" (and therefore to PLASMA) – but by then it is already too late to secure the request. A *PlugIn* is another possibility for a browser to pass data to another program and to read back the response from this program. The PlugIn API is a proprietary extension of the Netscape Navigator API and is not supported by other browsers.

The PLASMA relevant data are embedded into the HTTP protocol; to fulfill the requirements for browser independent communications there is to find a way to pass the information onto PLASMA on the way from client to server and conversely, i.e. to filter the data for the security platform. Therefore it is not possible to use a special propose filter module because HTTP is not aware of the filter.

The proxy, however, is well known of the HTTP protocol. Proxies in W3 buffering the informations from the server on clientside and handing over these data to the browser if they are completely arrived. So the proxy is the component where the filter functionality can be integrated. Therefore PLASMA has been integrated into the World Wide Web using a proxy on the client side and using CGI programs on the server side. This allows a secure usage of the World Wide Web by all clients, no matter what the browser of their choice might be (Netscape, Mosaic,...).

[7] The certificate structures used in SecuDe comply with the X.509 authentication framework [15] which requires the existence of certification authorities for the certification of the asymmetric public keys.

[8] The Mosaic browser family offers a feature to access programs such as security platforms directly from the web client. To achieve this goal, the CCI was defined for the client side and on the server side the CGI specification was established.

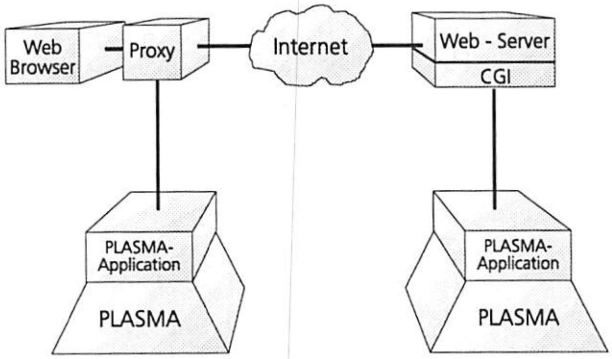


Figure 3: Overview of the implemented proxy architecture

The data to be passed on to PLASMA can only be transferred on WWW via the HTTP protocol. Informations intended for PLASMA are so-called PLASMA tokens. There exists three different kinds of PLASMA token; the PLASMA application has to discern whether the token is an authentication token (type `X509`), a document or *Container* token (type `Cont`), or a *Final* token (type `Finl`). It must be possible in any phase of the protocol between the PLASMA applications to pass data from the WWW layer onto PLASMA:

- In the authentication phase PLASMA on client side must be able to retrieve the requests for embedding the relevant PLASMA token into the PLASMA packet.
- In the document transmission phase cryptographic operations must be performed on the informations regarding which document the client or user requested; therefore all hyperlinks of the previously sent pages must be embedded in PLASMA packets.
- On the server side the CGI programs detect in the requests the embedded PLASMA tokens so they may be passed on to the PLASMA application.
- Subsequently they retrieve a HTML page including a hyperlink to the next CGI program in sequence as well as the corresponding PLASMA token; this token must then be passed to the PLASMA application on the client side.

Because these PLASMA tokens can only transferred on the WWW, they are transferred connected to elements of the HTTP protocol, i.e. the replies of the server and the requests of the clients. These HTTP elements include comments field not considered by the HTTP protocol which can be used to fill in the identification tag for the filtering process. These packages will be called in the following as PLASMA packets. PLASMA packets contain consequently an identification tag which the CGI programs and the proxy may detect.

The communication between web client and web server is not diverging from the normal HTTP protocol, therefore the architecture presented here can also be performed with a firewall. It is, however, important to note that the proxy used as a filter should not be the same proxy used for protecting a domain by a firewall. For a secure web scenario every client user has to start the proxy on this machine, where he started his own PLASMA system; every user should start in the best case his own proxy.

4.1 The components of the proxy architecture

The following section deals with the components of the secure World Wide Web architecture using

PLASMA and a proxy and describes their responsibilities. The separate components are PLASMA itself as a server process on the client and server side, an application of PLASMA on both sides which is capable of calling the application independent interface functions, the proxy on the client side and the CGI programs on the server side which transfer the data from the web server in their defined states to PLASMA. An overview of the components used in the proxy architecture is given in Figure 3.

PLASMA as a server process The "philosophy" of the World Wide Web can be summed up as follows: a client establishes a channel to a WWW server using an URL request. The server responds to this request by sending the requested information back to the client (if possible) and closing the communications channel immediately thereafter. If the client wants to direct another request to the server, a new communications channel must therefore be established.

World Wide Web servers can call external programs via the CGI, for example to perform database accesses or to activate security software. The CGI specification uses a simple scheme for such calls; the CGI program must terminate once it has processed the data and returned it to the web server. The server in turn passes the data generated by the CGI program on to the client and "forgets" everything about the transaction. Under normal circumstances this should not be a problem – unless a CGI program needs to maintain state between two calls.

If one wants to use PLASMA to secure web communications on the server side, this means that, for example, the server sends the request for the first authentication token to PLASMA via a CGI call. After serving this request, the CGI program terminates, therefore PLASMA terminates as well. All data that would have to be stored to maintain a secure link between client and server are then lost (this includes public keys and certificates as well as session keys etc.).

Therefore PLASMA must run permanently on the server side and it must not terminate once the CGI application terminates when returning data to the web server. The PLASMA system is therefore turned into a daemon process; this allows it to maintain state information for several communications links across several CGI calls within PLASMA[9]. Similar considerations apply to the client side; since the proxy also maintains only transient processes for each request, the state information for a secure communications link must be maintained in a PLASMA application as well.

The CGI programs As has been mentioned before, World Wide Web servers are able to adress external programs via the CGI. The call of a CGI program is usually initiated by activating a hyperlink that was found in a HTML document previously transferred to the client side. Since the activities of the CGI programs are well defined within the web scenario for each state of the protocol between sender and receiver, separate CGI programs were defined for each state.

Both CGI calls during the authentication phase are performing identical tasks with regard to PLASMA, they do, however, send different web pages as a response to the client.

The CGI program called during document transfer gets the requested document and decides whether or not the client has access to the requested document by means of the embedded client *DName*[10]. Lastly, there is a separate CGI program which gets called if a request for terminating the *session* with the client is detected.

All CGI programs are searching for the PLASMA identification string and passing as the proxy on client side these data onto PLASMA.

[9] PLASMA on both sides is capable of securing several simultaneous communications, so a web server is able to serve several clients simultaneously and similarly a client is capable of starting several simultaneous requests to different servers using secure connections.

[10] *DNames* are unique identifiers of the participating parties which are defined in the X.500 standards suite (cf. [15]).

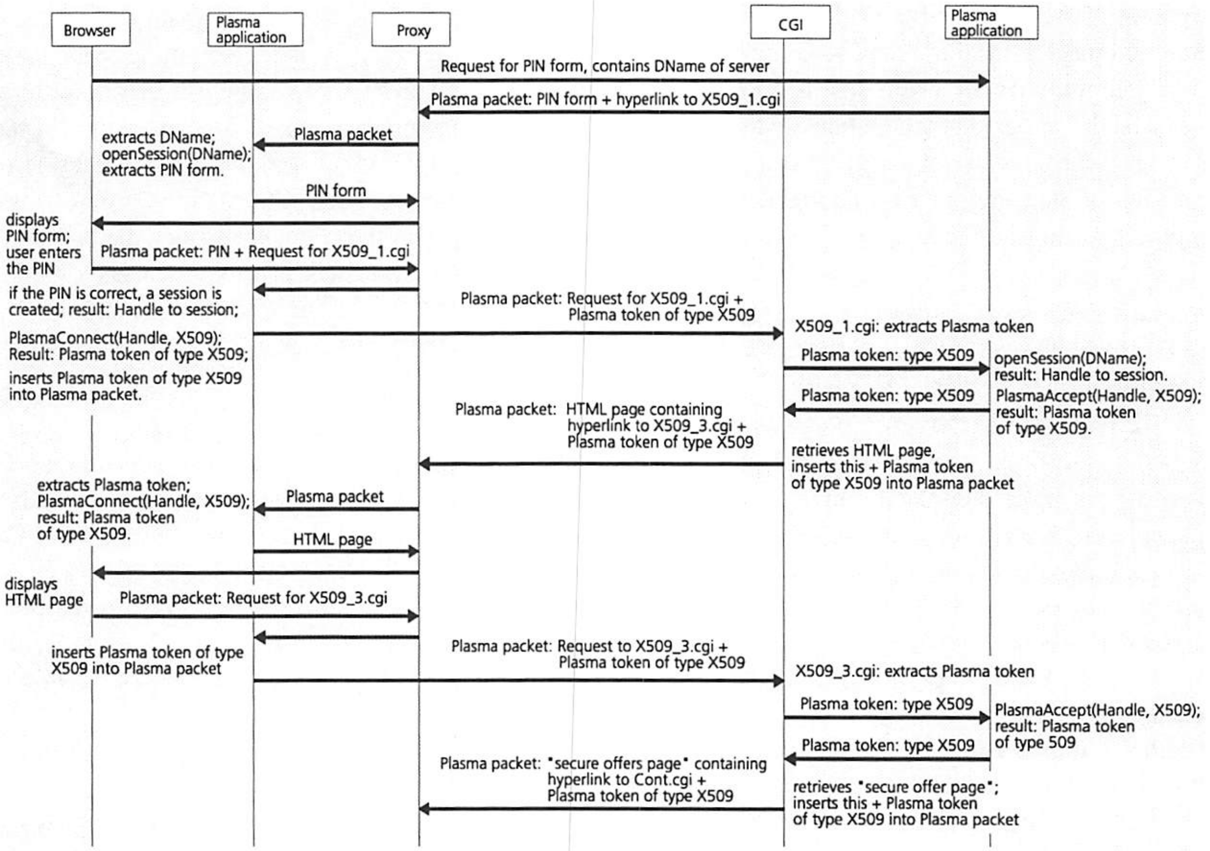

Browser
Plasma application
Proxy
CGI
Plasma application
Request for PIN form, contains DName of server
Plasma packet: PIN form + hyperlink to X509_1.cgi
extracts DName; openSession(DName); extracts PIN form.
Plasma packet
PIN form
displays PIN form; user enters the PIN
Plasma packet: PIN + Request for X509_1.cgi
if the PIN is correct, a session is created; result: Handle to session;
PlasmaConnect(Handle, X509); Result: Plasma token of type X509;
inserts Plasma token of type X509 into Plasma packet.
Plasma packet: Request for X509_1.cgi + Plasma token of type X509
X509_1.cgi: extracts Plasma token
Plasma token: type X509
openSession(DName); result: Handle to session.
Plasma token: type X509
PlasmaAccept(Handle, X509); result: Plasma token of type X509.
Plasma packet: HTML page containing hyperlink to X509_3.cgi + Plasma token of type X509
retrieves HTML page, inserts this + Plasma token of type X509 into Plasma packet
extracts Plasma token; PlasmaConnect(Handle, X509); result: Plasma token of type X509.
Plasma packet
HTML page
displays HTML page
Plasma packet: Request for X509_3.cgi
inserts Plasma token of type X509 into Plasma packet
Plasma packet: Request to X509_3.cgi + Plasma token of type X509
X509_3.cgi: extracts Plasma token
Plasma token: type X509
PlasmaAccept(Handle, X509); result: Plasma token of type 509
Plasma token: type X509
Plasma packet: "secure offers page" containing hyperlink to Cont.cgi + Plasma token of type X509
retrieves "secure offer page"; inserts this + Plasma token of type X509 into Plasma packet

Figure 4: The interactions in the authentication phase

The proxy From the security perspective the proxy is merely a filter deciding whether or not data must be passed through PLASMA in this architecture; it searches for a PLASMA identification tag in each data packet it receives. Data containing the PLASMA identification tag are passed on to PLASMA, otherwise the proxy passes the data on to the browser or the server, respectively, without interacting with PLASMA.

In order to maintain the full functionality of a "normal" proxy while embedding the necessary functionality, a CERN HTTPD 3.0 was used as a base for embedding the PLASMA modifications. Porting these modifications to other proxy architectures should be fairly easy since these are well embedded. The drawback inherent in this approach is the necessity to follow standard operating procedures, i.e. each browser request must be followed by a response from the server.

Since the browser knows nothing of the security enhancement of the communications channel, all transactions must be performed between the proxy and the server. Each request for PLASMA services must contain such a string which allows the proxy to intercept it and send it to the PLASMA application; the same goes for incoming data from the server; therefore the identification string is inserted by the CGI program.

The PLASMA applications on client and server side The PLASMA applications are the agents tasked with accepting transferred data from the proxy on the client side and from CGI programs on the server side. It passes these data on to PLASMA and returns the output to the proxy or to the CGI programs for secure transfer across the public network. The PLASMA application gets passed only such data packets which actually have to be modified cryptographically by PLASMA in either the "to" or "from" direction.

It is the responsibility of the PLASMA application to analyze received data packets and to pass these PLASMA tokens on to the security platform using the correct interface functions of the application independent interface [8].

Since PLASMA is required to operate permanently as a background process, socket connections between the proxy and the PLASMA application on the client side as well as between the CGI programs and the PLASMA server application are necessary.

4.2 The dynamic model

The following section details the interactions between the separate components required for secure web communications. The dynamic model is subdivided into three phases: the authentication phase, the secure transmission phase and the connection teardown phase.

The authentication phase The authentication phase commences by the client establishing a secured connection with the server within PLASMA; to achieve this the application on client side calls the PLASMA C–API function `openSession()`. For this the *DName* of the server is required which is inquired in the first request. The server's response contains the *DName* as well as a PIN form into which the user at the client side will enter his PIN.

Once the `Session` has been opened successfully, the client calls the authentication function for the sender side for the first time (`PlasmaConnect()`). The result is a PLASMA token of the type `X509` which is then sent to the server. Upon receiving this PLASMA token (`X509`) at the server side, a *session* is created within PLASMA and the authentication towards the client is continued (first call of the authentication for the receiver side (`PlasmaAccept()`).

The exact flow of the protocol for authentication may be gleaned from Figure 4. In this phase first the PIN form including the server *DName* will be transmitted to the client. Next the CGI programs `X509_1.cgi` and `X509_3.cgi` are called on the server side, which are sending different HTML pages to the client side; the first HTML page is only used for sending the hyperlink to the next CGI program (`X509_3.cgi`), the next page already present the offers of the server.

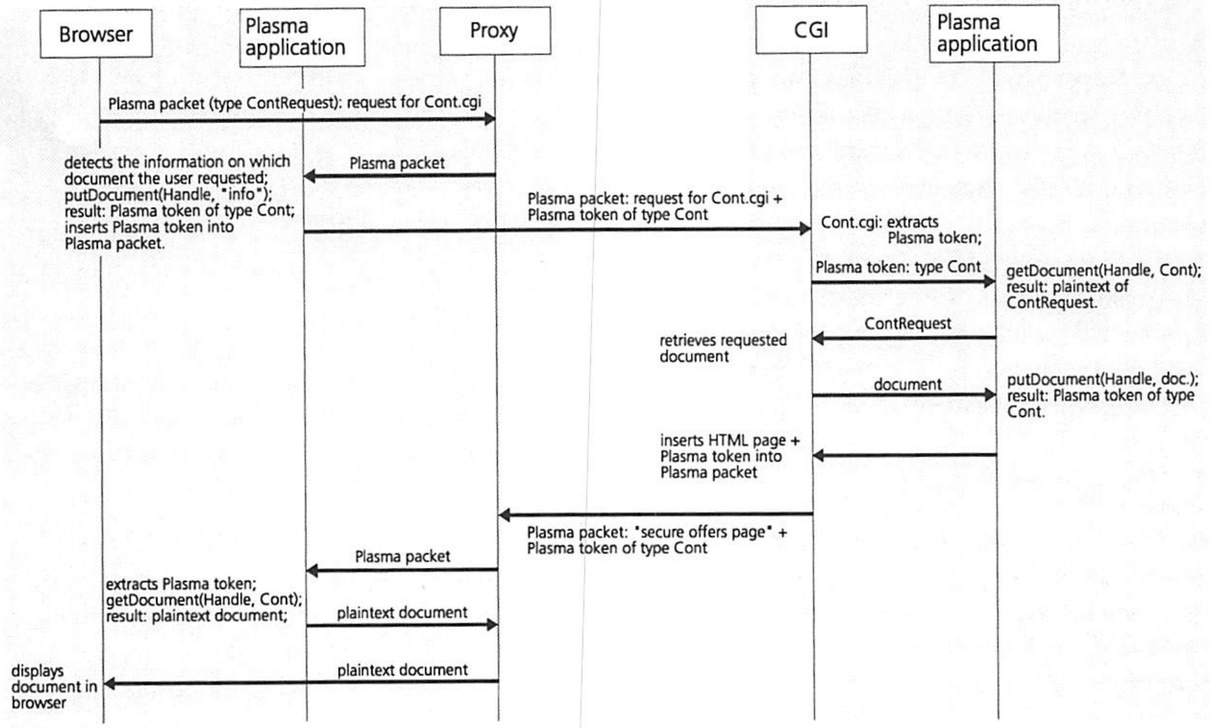
In this state only PLASMA token of type `X509` are transmitted – both applications have appropriate knowledge as to which API calls need to be performed when receiving PLASMA tokens of this type.

The document transmission phase After a successful three-way-authentication requests and replies between the web client and server may be transmitted securely. The server offers its documents via the *secure offers page* (cf. Figure 5); on the client side the user selects a document and activates the corresponding CGI program upon confirmation. The call of the CGI program is not encrypted; however, the information regarding which document has been requested by the user must be encrypted at the client side[11].

In this phase only tokens of the type `Cont` are transmitted; the PLASMA application is wired to call the filter function `getDocument()` for reverting the cryptographic operations on this token type. In this phase plaintext data are passed onto PLASMA for cryptographic operations using the `putDocument()` API function.

The CGI program `Cont.cgi` gets called on the server side. It decrypts the information on the requested document and encrypts the document itself prior to transmission. The HTML pages sent in this phase represent the documents or offers of the server; the exact protocol is shown in Figure 5.

[11] This requires the application on the client side to detect the condition that the API function `putDocument()` for cryptographic operations in "to" direction must be called, therefore the request type *ContRequest*, which is also a PLASMA packet, was introduced.


Browser
Plasma application
Proxy
CGI
Plasma application
Plasma packet (type ContRequest): request for Cont.cgi
detects the information on which document the user requested; putDocument(Handle, "info"); result: Plasma token of type Cont; inserts Plasma token into Plasma packet.
Plasma packet
Plasma packet: request for Cont.cgi + Plasma token of type Cont
Cont.cgi: extracts Plasma token;
Plasma token: type Cont
getDocument(Handle, Cont); result: plaintext of ContRequest.
ContRequest
retrieves requested document
document
putDocument(Handle, doc.); result: Plasma token of type Cont.
inserts HTML page + Plasma token into Plasma packet
Plasma packet: "secure offers page" + Plasma token of type Cont
Plasma packet
extracts Plasma token; getDocument(Handle, Cont); result: plaintext document;
plaintext document
plaintext document
displays document in browser

putDocument(Handle, Data,...) := plaintext data are passed on to Plasma for cryptographic operations,

getDocument(Handle, Data,...) := encrypted data are passed on to Plasma for reconstruction;

ContRequest: this Plasma packet contains the call of the CGI program Cont.cgi as well as information on which document the client requested.

Figure 5: Interactions in the data transmission phase

Connection teardown All HTML pages of a server allowing secured communcations using PLASMA which offer documents contain a hyperlink to the CGI program `Finl.cgi` – refer to Figure 6 at the bottom. Upon activation of this hyperlink the *session* at the client side is torn down by calling the API function `closeSession()` of PLASMA[12].

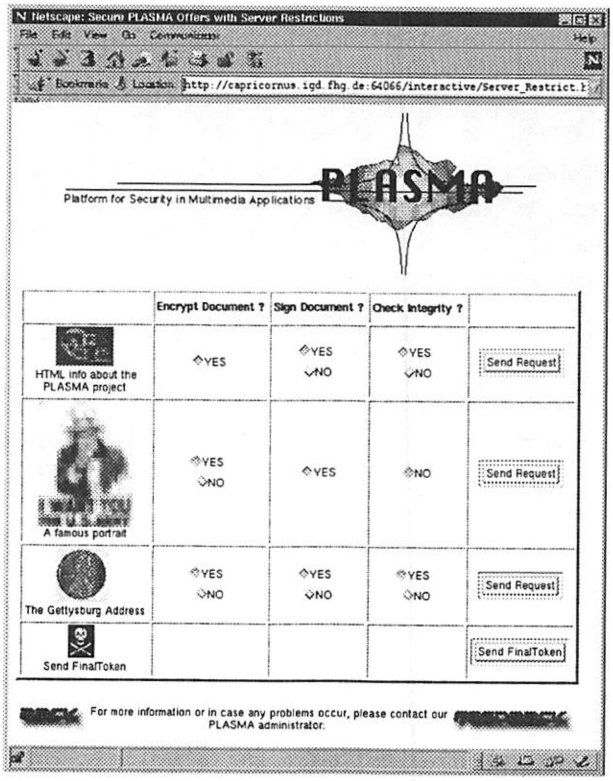
The result of this connection teardown is a PLASMA token of the type `Finl` which now must be sent to the server. The application on the server side also recognizes that the *session* is to be disconnected upon receiving this token of type `Finl` and thus also calls `closeSession()`.

5 Summary

In conclusion the results of the integration of the PLASMA security platform into the World Wide Web are summed up — these results are made possible by the implementation of the idea of *high level security* within PLASMA.

The just presented architecture allows a mutual authentication of web clients and servers which may also be achieved by means of other security platforms; after a successful authentication the server knows which client user wants to obtain data from him and may use this identification by means of appropriate CGI programs to grant or deny access (access control to the requested documents).

[12] This requires the application on the client side to detect the condition that the API function for conncection shutdown must be called, therefore the request type *FinlRequest* was introduced, also a PLASMA packet.


File Edit View Go Communicator Help
Bookmarks Location
Platform for Security in Multimedia Applications
PLASMA

Netscape: Secure PLASMA Offers with Server Restrictions

http://capricornus.igd.fhg.de:64066/interactive/Server_Restrict.h

	Encrypt Document ?	Sign Document ?	Check Integrity ?	
HTML info about the PLASMA project	◇YES	◇YES ◇NO	◇YES ◇NO	Send Request
A famous portrait	◇YES ◇NO	◇YES	◇NO	Send Request
The Gettysburg Address	◇YES ◇NO	◇YES ◇NO	◇YES ◇NO	Send Request
Send FinalToken				Send FinalToken

For more information or in case any problems occur, please contact our PLASMA administrator.

Figure 6: Interaction opportunities for the user in the web demo

Requests and responses may now be secured — by locating the security platform close to the application the interactions of the user may be considered, i.e. the server can offer its documents in such a way as to allow the client to determine whether the document is to be sent confidentially, non-reputiably or with integrity checks. Furthermore the server may predetermine which documents need to be protected using specific security services; the web page in Figure 6 exemplifies the opportunities for interaction with the user; the second image from the top may be treated confidentially and be transmitted non-reputiably; interaction with the user is possible since the generic security services within the platform have been realized as independent modules – these merely have to be activated by a mouse click via the interface of PLASMA.

Finally, and this is the most significant feature facilitated by using PLASMA, the different media and different structures of the HTML documents may be differentiated cryptographically. Using PLASMA it is possible to ensure that forms, which may be containing a contract, will always be signed – both by the server to make sure that the client knows he is transferring his credit card number to an authorized server as well as by the client side to make sure that the client user has actually signed the "contract" in that particular form; using PLASMA it is possible that for example textual data is always integrity protected and images are protected against loss of confidentiality during transmission.
Finally it ensures that an unauthorized disruption of a session by a client or server can be detected.

6 Acknowledgements

I would like to thank everyone who has contributed to the design of the just presented architecture for their insights and useful discussions; in particular I would like to thank Xiaodong Liu for the invaluable support in evaluating peculiar problems of the WWW and for aid in the specification of the secure WWW architecture. I would like to thank Dr. Eckhard Koch for the opportunity to realize this project. Furthermore I would like to extend particular gratitude to Xiaodong Liu again, Henning Daum, Stephen Wolthusen and Ingo Jankowski for extremely productive assistance in implementing these system, as well as Dr. Christoph Busch for proofreading.

A different approach for implementing the web security scenario using the CCI system with a Mosaic browser and a different underlying security technology platform was developed by Cheng Dong, Mehrdad Jalali and Jian Zhao, also at the Fraunhofer-Institute; this system has provided several crucial insights into the particular problems one encounters when securing the web.

References

[1] T. Berners-Lee, A. Luotonen, H. F. Nielsen, A. Secret (1994) The World–Wide–Web. *Communications of the ACM, Vol.37 No. 8.*

[2] H. Cheng, X. Li (1996) On the application of image decomposition to image compression and encryption. *Chapman & Hall, Communications and Multimedia Security II, ed. P. Horster, 116-127.*

[3] B. Fernandez, R. Nair, M. Larrondo-Petrie, Y. Xu (1996) High–Level Security Issuses in Multimedia/ Hypertext Systems. *Chapman & Hall, Communications and Multimedia Security II, ed. P. Horster, 13-24.*

[4] R. Fielding, J. Gettys, J. Mogul (1996) Hypertext Transfer Protocol – HTTP/1.1. *IETF draft.*

[5] A. O. Freier, P. Karlton, P. C. Kocher (1996) The SSL Protocol Version 3.0. *Netscape Communications Corporation.*

[6] M. Gehrke, E. Koch (1992) A Security Platform for Future Telecommunication Applications and Services. *Proc. of the 6th Joint European Networking Conference.*

[7] A. Krannig (1996) PLASMA - Platform for Secure Multimedia Applications. *Chapman & Hall, Communications and Multimedia Security II, ed. P. Horster.*

[8] A. Krannig, H. Daum (1996) PLASMA — The Application Independent API. *Fraunhofer–IGD Darmstadt, Technical Report.*

[9] Linn, J. (1993) *RFC 1508 – Generic Security Service Application Programming Interface*

[10] P. Lipp, v. Hassler (1996) Security concepts for WWW. *Chapman & Hall, Communications and Multimedia Security II, ed. P. Horster.*

[11] NCSA (1996) CCI Specification. *http://www.ncsa.uiuc.edu/SDG/Software/XMosaic/CCI/cci-spec.html.*

[12] NCSA (1996) CGI The Common Gateway Interface. *http://hoohoo.ncsa.uiuc.edu/cgi/.*

[13] H. Reif (Juni 1997) Secure Socket Layer: Chiffrieren und Zertifizieren mit SSLeay. *IX Multiuser Multitasking Magazin.*

[14] E. Rescorla, A. Schiffman (1996) The Secure HyperText Transfer Protocol. *IETF draft.*

[15] B. Schneier (1996) Applied Cryptography, 2nd ed. *Wiley.*

[16] W. Schneider (1993) SecuDe: Overview. *GMD–TKT Darmstadt.*

[17] Wray, J. (1993) *RFC 1509 – Generic Security Service API : C Bindings*

[18] P. Zimmermann et al. (1993) PGP. *Phil Zimmermann.*

Security of Web Browser Scripting Languages: Vulnerabilities, Attacks, and Remedies

Vinod Anupam Alain Mayer
Bell Laboratories, Lucent Technologies
600 Mountain Avenue
Murray Hill, NJ 07974
{anupam,alain}@bell-labs.com

Abstract

While conducting a security analysis of JavaScript and VBScript, the most popular scripting languages on the Web, we found some serious flaws. Motivated by this outcome, we propose steps towards a sound definition and design of a security framework for scripting languages on the Web. We show that if such a security framework had been integrated into the respective scripting languages from the very beginning, the probability of preventing the multiple security flaws, that we and other research groups identified, would have been greatly increased.

1 Introduction

JavaScript and VBScript are popular scripting languages used for Web-page design. A user accessing a JavaScript/VBScript enhanced Web-page causes scripts to be downloaded onto the user's machine and to be executed by the interpreter of the user's browser. Scripts typically cannot (directly) access the user's file-system or the network. This is probably the reason that, in contrast to the Java programming language, no formal security model and hence no explicit rules were ever documented on what is and what is not allowed by scripts. A string of serious security flaws discovered by several research groups, including successful attacks on patches issued to fix original flaws, shows that this is a dangerous omission. In particular, we found flaws which allow private data supplied by a user (e.g., credit card numbers, passwords, e-mail address, etc) in a Web transaction to be captured by an attacker. Such an intrusion works even when a user employs encryption (e.g., SSL), since the data is captured either before it is encrypted or after it is decrypted. In contrast to some of the security flaws found in Java (see [MF97]), the vulnerability we discovered does not lead to full system penetration where an attacker can access a user's resources (files, processes) at will. It might thus be argued that such flaws are less serious. However, security and privacy concerns (see, e.g., cover story of *Time Magazine*, dated 8/15/97) have been the single most important barrier to electronic commerce achieving its multi-billion dollar potential. In this light, attacks on a user's security and privacy, are a matter of serious concern.

Motivated by the above considerations, we proceed to show necessary steps towards a general security framework for scripting languages on the Web. We put forward the notion of a *safe interpreter*. A safe interpreter must assure:

- *Data Security.* Data provided by the user (possibly encrypted before it is transmitted) can only be accessed by the intended recipient; the possibility that credit-card data, transaction details, etc. may be obtained by someone else is highly damaging.
- *User Privacy.* Information about the user should not be given out unless explicitly allowed by the user; this protects against unwanted tracking, identification dossiers, junk e-mail, surreptitious file uploads, etc.

As a concrete example, we give excerpts from *Secure JS*, our proposal for a more secure version of JavaScript. In this paper, our treatment covers security issues of scripting languages up to Navigator 4.* and IE 4.*. However, we do not address code signing, a topic that transcends scripting languages. Also, today's browser environment allows embedded scripts to communicate with entities like applets and plug-ins, outside the scripting languages proper, adding a lot to the scope of what scripts can do via these entities. This "composition" of different entities with different security policies and frameworks is not well understood; other safe scripting environments, such as Safe-Tcl (see [B94, OLW96]), do not allow such composition. We can only touch upon this issue; a thorough treatment is beyond the scope of this paper.

The notion of a secure operating system, which provides safe containers for different Web technologies (Java, plug-ins, JavaScript) would help in achieving the above goals. However, we emphasize that most of the uncovered vulnerabilities of scripting languages originate within the language itself. For instance, a rogue site may contain attack scripts which can access (and send back) data obtained from other documents without any penetration of the underlying operating system or concurrent application.

Finally, we show that if our framework had been designed and integrated when either JavaScript or VBScript were conceived, the probability of preventing the string of security flaws, that were identified by us and other research groups, would have been greatly increased. This motivates the need for future designs of scripting languages to explicitly consider security aspects during initial design. However, security is never absolute. Implementation errors, even in a sound security design, are not unlikely and can be exploited by an attacker. Such flaws, however, are harder to identify by an attacker.

Organization of the Paper: In Section 2 we review the basic concepts of browser scripting, and briefly introduce JavaScript and VBScript, the two most popular scripting languages for Web browsers. Section 3 presents the essence of our attacks on JavaScript and VBScript capable browsers. We then propose, in Section 4, a security framework for scripting languages on the Web. We illustrate this framework with concrete notions and examples taken from our proposal of *Secure JS*. In Section 5, we discuss our attack in more detail and point out how our framework would have helped in preventing it. Section 6 concludes. In the Appendix, we discuss attacks designed by other groups and again show how our framework would have helped in preventing them.

2 Browser Scripting Languages: An Overview

JavaScript (see [F97, KK97]) is a simple procedural language that is interpreted by Web browsers from Netscape Corp. (*JScript*, Microsoft Corp.'s implementation, is a clone that is interpreted in Microsoft's Web browsers. In the rest of this paper, we use JavaScript to refer to both strains.) JavaScript is object-based in the sense that it uses built-in and user defined extensible objects, but there are no classes or inheritance. The code is integrated with, and embedded in, HTML. By default, JavaScript provides an object-instance hierarchy that models the browser window and some browser state information. E.g., the *navigator* object provides information about the browser to a script, and the *history* object represents the browsing history in the browser window. Also, through a process called 'reflection', JavaScript automatically creates an object-instance hierarchy of elements of the script's HTML document when it is loaded by the browser. The *location* object represents the URL of the current document, while the *document* object encapsulates HTML elements (forms, links, anchors, images etc.) of the current document. This defines a unique *name space* for each HTML page and thus for each collection of scripts embedded in that page. Variable data types are not declared, i.e., *loose typing*. Object references are checked at runtime, i.e., *dynamic binding*.

VBScript (see, e.g., [L97]) is an application scripting language that looks a lot like Visual Basic. It is loosely typed and object based. It can be used for scripting Microsoft's browser, Internet Explorer, with which it communicates using ActiveX Scripting. ActiveX Scripting allows host applications, like browsers, to compile and execute scripts, and manage the name-space exposed to the script. The ActiveX Scripting Object Model (SOM) creates an object instance hierarchy containing, among other things, *window*, *navigator* and *history* objects as described above. Also, the *location* object represents the current URL, and the *document* object reflects the current HTML document. VBScript scripts are embedded in HTML documents, and are interpreted automatically when the document is loaded.

3 Our Attack

While conducting a security analysis of JavaScript we discovered a serious vulnerability in JavaScript-capable browsers. We subsequently analyzed VBScript, and discovered that the vulnerability could be exploited equally effectively using VBScript. The vulnerability in Microsoft browsers was thus also in the ActiveX Scripting Object Model - JScript and

VBScript interpreters are front ends that communicate with the underlying ActiveX SOM objects.

3.1 Overview

On the Web, scripts embedded in multiple browser windows containing documents from the same Web site (same domain name) are allowed to access data in each other, in order to support multi-windowed user interfaces. Our analysis revealed, however, that browser windows could be tricked into trusting attack scripts from rogue sites, thus allowing them to access their data. A rogue site could be set up to track all Web-related activity of visitors even after they had left the site, using a Trojan-horse attack. The tracking provided access to all data typed into forms, including password fields, cookies, and visited URLs. The data was extracted right in the browser, so using a secure encrypted connection to retrieve documents didn't accord the user any extra protection. Likewise for users behind firewalls - data was intercepted while in the browser, and transmitted to the outside rogue site via a proxy server.

3.2 Implications

This browser vulnerability has a serious implication for Web users. Once infected by the Trojan horse, the user's Web interaction is fully exposed to the attacker - every URL retrieved, all data typed into forms - including credit card numbers and passwords, all cookies set by servers accessed etc.

The HTTP protocol supports a facility for authenticating Web users. Many Web-based services however use alternate methods of authorization that provide more flexibility. These methods involve the use of dynamically generated, opaque "session keys" embedded in URLs, in hidden fields of forms or in cookies. The ability of the attack to access such information in an HTML document makes all of these authentication mechanisms susceptible to compromise.

This browser vulnerability also has a serious implication for intranets. Most users use the same browser to access information on the intranet as well as the Internet. A user who has been "attacked" using this vulnerability has essentially compromised the firewall for the duration of the browsing session - the Trojan horse is able to extract data from subsequently loaded intranet documents and transmit it to an external entity. Any data that the user enters into forms - ID numbers, vendors and prices, bug reports, passwords and other proprietary information can be relayed to the outside. In addition, the document itself can be subjected to analysis, yielding information about different fields, options and their corresponding values. Particularly serious is the fact that this exploit bypasses security provided by secure sockets or HTTP authentication, since many Web servers and other business platforms are administered over the Web nowadays. Information about document URLs and links on pages can also provide the attacker with an idea of how the domain is organized. This knowledge may be used to help conduct other attacks. One can envisage a scenario where the Trojan horse actively browses the intranet, possibly under remote control from outside the firewall, trolling for any and all information accessible via the browser.

4 The Notion of a Safe Interpreter

4.1 Execution Environment

We consider a networked system. The basic entities on a single machine in our execution environment are *windows*, *contexts*, *scripts*, *interpreters*, *name-spaces*, and *external interfaces*. A concrete example of a *window* is a browser window. A user can have multiple windows open on her machine at any point in time. A window and its content (e.g., the current HTML document displayed) define a *context*. When the content of a window changes (e.g., as the result of loading a new HTML document), the window's context changes as well. A *script* is a code fragment in a scripting language embedded in the content of a window. A document may have multiple code fragments embedded in it. All of them share the window's context. The content of a window (and hence any embedded script) is typically downloaded from a Web server across the network by the browser. An embedded script is executed within the window's context by an *interpreter*. A *name-space* is the hierarchy of objects accessible to a script executing within the window's context. A script might also be able to invoke actions external to its context via an *external interface*. For example, a script can cause the browser to open HTTP/FTP connections, send e-mail etc. Finally, a script can establish a *trust relationship* between its context and another window's context, which enables it to access the name-space of the other context via a reference. For example, Figure 1 depicts three open windows (w, w', and w'') on a user machine, where the scripts executing in w's and w''s context can access each other's name-space and scripts in w can access some external interface, giving them additional capabilities.

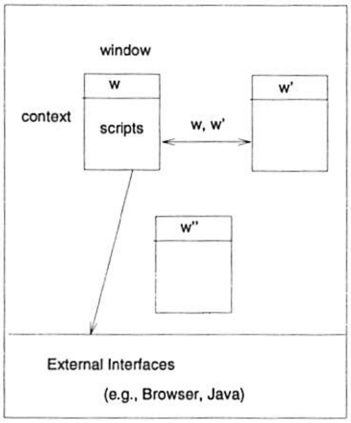


Figure 1: Execution Environment

In some instances, a browser window may contain a framed document with multiple *frames*, which are subwindows containing other documents. From the perspective of a safe interpreter, each frame in a framed document is an independent *window*.

4.2 Safe Interpreter

A script loaded into a window in the above execution environment originates at an arbitrary machine on the network, and is therefore an *untrusted* entity on the user's machine. Hence, a safe interpreter has the task of isolating scripts from executing any unsafe commands (those that could result in security compromises if misused), thus implementing what is called a *padded cell* in [OLW96]. The interpreter has to implement access control with respect to objects within the script's own context. Objects containing browser or window data for example, should only be read-accessible. A safe interpreter has to isolate contexts from each other: a user might decide to provide some information in one context to be sent back to a specific machine - this information should not be accessible to any other machine on the network, and hence must be inaccessible to a script in a different context. On the other hand, under certain circumstances, scripts in different contexts require mutual access, e.g., an application on the user's machine which runs in multiple windows requires the ability to access data in different windows. A safe interpreter must allow this kind of access to scripts in trusted contexts

In summary, a safe interpreter has to implement *access control*, *independence of contexts*, and *management of trust* among different contexts. Provision for these components does not realize a particular security policy. Rather, it gives a *framework* in which a variety of security policies can be easily implemented. We discuss these components in the next few sections.

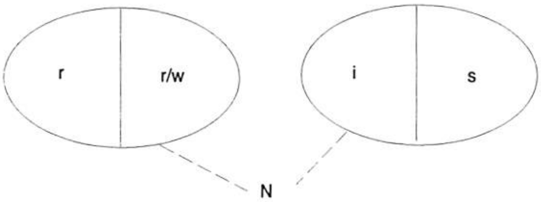


Figure 2: Partitioning of the Name-space

4.3 Access Control

The first task of a safe interpreter is to clearly specify what objects are accessible to an arbitrary context - i.e. what constitutes its name-space. A safe interpreter (1) defines the name-space N_C of a context C, i.e., which objects exist at the time C is activated, (2) the initial values of these objects, and (3) ensures and implements the following partitioning of N_C:

- N_C^r: Items in the object-instance hierarchy readable by a script in C.
- N_C^w: Items in the object-instance hierarchy readable and writable by a script in C.
- N_C^s: Items created by a script in C.
- N_C^i: Items created by the interpreter in C.

We use the term *item* to describe either an object, function, variable, or an object property. Functions do not need to be treated separately in this framework - they are executable if they are readable. Figure 2 shows the following invariants: N_C^r and N_C^w are disjoint, N_C^s and N_C^i are disjoint. Also $(N_C^r \cup N_C^w) = (N_C^s \cup N_C^i) = N_C$. N_C^i is special in the sense that insertion and deletion of elements into N_C^i occurs only when the interpreter creates and deletes a context - scripts in an active context cannot directly add and remove items from this set. Initial values of some of the items in N_C^i depend on the window, in which the context is loaded and activated - the execution environment. It is the role of the safe interpreter to assure correct initialization.

Design Issue 1 for Secure JS: Defining the Name-space The first design issue is to exactly define $N_C = (N_C^s \cup N_C^i)$. The client-side JavaScript environment is defined by its *object instance hierarchy* - objects have *properties* which may be other objects. N_C^i is roughly equivalent to the entire

JavaScript *object hierarchy* at the time an HTML document is being loaded into a window. Every object created by a script after the document is already loaded belongs to N_C^s. The `window` object is the root of the JavaScript object instance hierarchy. It has a number of properties. For example, `window.name` is a string that contains the browser-window's name. The `window` object also contains other properties such as the `window.document` object that contains HTML elements reflected from the current document, the `window.navigator` object that encapsulates properties of the browser software, the `window.history` object that represents the browsing history in that window etc. All of these properties are created by the interpreter when a context is loaded and hence belong to N_C^i. In contrast, consider the code fragment `var foo; foo = "bar";`. This assignment is equivalent to `window.foo = "bar";`, and results in the `window` object getting a new property `foo`, whose value is the string `"bar"`. This property `foo` of `window` is created by a script after the context has been loaded. Consequently, `foo` belongs to N_C^s.

Explicit specification of what belongs to the name-space regulates what can ever directly be accessed by a script. A more fine-grained specification of what belongs to N_C^i depends on the policy for *User Privacy*. For example, `document.referrer` contains the URL of the document from which the user reached the current document. Some of the properties of `window.navigator` (e.g. *userAgent*) contain information about the user's operating system. Many privacy-conscious users employ privacy proxies, such as the Anonymizer (see [Anon]) or LPWA (see [LPWA]) to filter this type of information from their HTTP requests. Hence, we recommend against the inclusion of these items in N_C^i (and hence in N_C). The `window.history` object contains an array of strings that specify URLs that have been previously visited by the user in that browser window. In all current versions of JavaScript, this array is **no longer** in N_C^i by default, to protect against unauthorized access to that information by scripts. Note that N_C^s contains objects created directly by scripts. From the standpoint of security and privacy, these objects do not need to be protected from the scripts that created them. This semantically splits the name-space into two disjoint subsets, one that contains objects to which scripts have largely unregulated access (N_C^s), and one that contains objects that can be accessed and manipulated only in specific ways (N_C^i.) □

Design Issue 2 for Secure JS: Read-only vs Writable Objects We now have to specify the partitions N_C^r and N_C^w. Semantically, certain properties of the name-space are unmodifiable, e.g., `document.lastModified` specifies the time at which the currently loaded document was last modified; `document.URL` contains a string that represents the URL from which the document was retrieved. Such objects are trivially in N_C^r, the read-only subset of the name-space. Other properties, such as `document.forms.length` etc., that are reflected from the loaded HTML document should not be directly modifiable by scripts, and hence belong in N_C^r. Object properties, like `document.domain`, are used for trust management (see Section 4.5) and should be in N_C^r. The `value` property of most form elements is intended to be modifiable by scripts, and thus belongs to N_C^w. However, the `value` property of a `FileUpload` element contains a user specified local filename to be transmitted over the network as part of the form submission. To disallow scripts from setting this property and retrieving arbitrary local files, this object rightly belongs in N_C^r. All objects created directly by a script are writable, and hence in N_C^w. One method of unauthorized activity involves a script covertly passing information to another script. This necessarily involves one script writing a value into an object that is subsequently read by another script. By treating writable objects (in N_C^w) more strictly, the interpreter can protect against establishment of such covert channels. □

We have seen how a safe interpreter assures that N_C is always disjoint from any private user resources on the machine, such as the file system, process data, sockets, etc. This is much like the concept of a padded cell in Safe-Tcl ([OLW96]). However, access to such resources is very useful. In Safe-Tcl, such accesses are regulated via a *master interpreter*, whose methods can be invoked by the safe interpreter. Hence, such access is regulated within the language itself. Unfortunately, for scripting languages on the Web, no such master interpreter is available. *External interfaces* give a script capabilities to invoke browser functionality, or methods of Java applets and ActiveX scripts. Each such invocation across interfaces needs to be examined by the interpreter, which has the option of (1) allowing the call to proceed, (2) aborting the call, or (3) asking the user's explicit permission for the call to proceed. However, the resulting overall security policy depends both on the policy of the safe interpreter

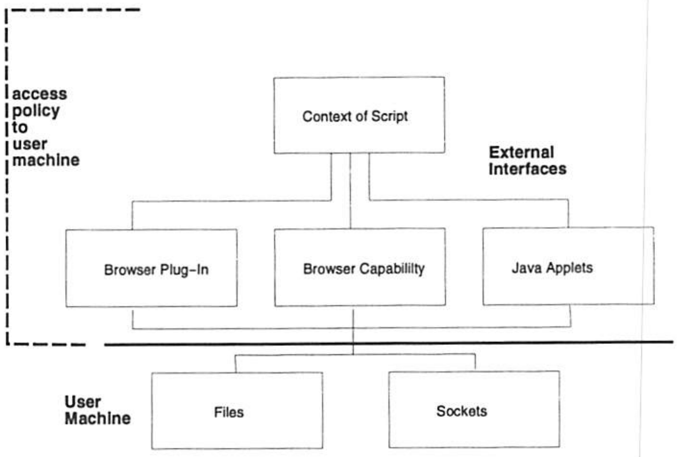


Figure 3: External Interfaces

and the policy of the external interface, the latter being beyond the control of the safe interpreter.
For example, Figure 3 presents a situation, for JavaScript, in which scripts can access the browser, Java applets, and browser plug-ins through external interfaces. As a result, a script can (albeit indirectly) access the file system or the network, making HTPP, FTP, and SMTP requests via the browser. Hence, the overall access security policy with respect to these resources depends, besides the safe interpreter, on the policies for the browser, for applets, and plug-ins. For example, if a new method is introduced for applets to access a file and the safe interpreter remains unchanged, then a script invoking this new method might not be subjected to an appropriate check by the safe interpreter. What is really needed, is sound semantics for *composition* of security policies. This is an area hardly investigated and understood at the time of this writing and is beyond the scope of this paper. A safe interpreter therefore should minimize the number of accepted external interfaces and their methods offered to scripts. Furthermore, it should adopt a conservative policy for every invocation of a call across an external interface, as well as external accesses to the script's name-space. A secure operating system providing safe containers and interfaces for different applications (Java, plug-ins, etc) is an alternative.

Design Issue 3 for Secure JS: Access to External Interfaces In order to be useful, JavaScript needs, at the very least, an interface to browser capabilities to access the network and special files. User supplied data (in forms) need to be transmitted back to an origin server. A script chooses the origin-server (URL) via `form.action` and the SUBMIT method via `form.method`, and submits the user data via `form.submit`. Whenever a protocol specified in the URL of the origin server is **not** HTTP, the safe interpreter should get a user's approval. SMTP and FTP requests potentially reveal a user's e-mail address. Furthermore, a user might be browsing via a privacy HTTP-proxy and thus unwittingly lose this protection when being switched to another protocol. The safe interpreter must prevent a script from directly accessing local files of a user. This includes any browser related files, such as bookmarks, caches, history, etc. A general treatment of external interfaces (with Java, plug-ins etc.) is beyond the scope of this paper - it requires an understanding of the very general issue of composition of security policies. □

4.4 Independence of Contexts

To ensure independence of different contexts, the following restrictions have to be enforced by a safe interpreter:

- For each active context C, N_C^w and N_C^s must be disjoint from $N_{C'}$ of each other context C'.

If an active context C is terminated and a new context C' is loaded and activated in a window w, a safe interpreter has to do the following operation in N_C in order to transform it into the initial name space $N_{C'}$ such that the name space $N_{C'}$ does not depend in any way on C (see also Figure 4):

- *Remove and rebuild N_C^i:* The set (instance hierarchy) is deleted and rebuilt according to the newly loaded HTML document. This process must ensure that while rebuilding, each item in N_C^w is set to a neutral value (null).
- *Delete N_C^s:* remove all its items.
- *Invalidate all references to N_C* within other contexts, which have been established by a trust relationship (see subsequent section).

Design Issue 4 for Secure JS: Independent Browser Windows A safe interpreter must ensure that the memory of the object hierarchy that corresponds to N_C^w is disjoint for different windows. This includes objects like the `window.navigator` object, which might represent the same browser instance for all windows. Enforcing disjointness protects against the establishment of covert channels between scripts. As described earlier, one method of unauthorized script activity involves a script covertly passing information to another script by

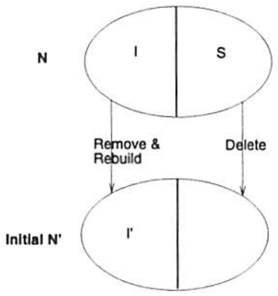


Figure 4: Independence of contexts loaded into same window

writing a value into an object that is subsequently read by the other script. This is particularly relevant for items whose value persists across document retrievals, e.g., some properties of the `window` object. By treating writable objects (in N_C^w) more strictly, the interpreter can protect against such covert channels. Secure JS does not allow values of items in N_C^w to persist when the current document is unloaded. For example, `window.name` is a string that specifies the name of the browser window. If scripts write a value into this property of the window, then any changes will disappear when the current document in the window gets unloaded. □

Design Issue 5 for Secure JS: Garbage Collection From the above discussion it follows that garbage collection based on simple *reference counting* is not sufficient to ensure security since it will not collect all objects that should be collected. Whenever a document is unloaded, the safe interpreter has to traverse the entire object hierarchy: Each item in N_C^s must be deleted. E.g., the code fragment `var i; i = 0` creates an item `window.i`, which must be removed. Appropriate items in N_C^i are deleted. Each remaining item in N_C^w must be set to a neutral value, even if it is above the `document` object in the hierarchy. E.g., `window.status` and `window.name`. All references contained in other windows that point to objects in the current window must be invalidated (see also next section), even if the current window remains open (and thus `window.closed` would be false, if the window reference remained valid). □

4.5 Management of Trust

While most contexts are typically independent, certain applications greatly benefit from establishing mutual trust relationships among contexts in different concurrently active windows. This allows, for example, the coordination of information in multiple windows when presented to a user. In order to maintain a secure system, we need to specify (1) when a script in context C is allowed to establish a relationship with context C' and (2) the subsequent privileges for $N_{C'}$ issued to scripts in C and vice versa.

If context C' (in window w') trusts C (in window w), then access to C''s name-space is allowed. The safe interpreter provides scripts in context C with a *reference* to context C'. Via this reference, a script in C can read all items in $N_{C'}^r$, read and write all items in $N_{C'}^w$ and insert items into $N_{C'}^s$. Note that the specification in Section 4.4 implies that if a script in C inserts an object into $N_{C'}^s$, this object will only be accessible to C as long as C' is active in w'. After that, C's reference to C' is set to null. We can extend this approach to different trust relationships of various degrees: (1) Read-only access to C''s name-space. (2) Access only to objects in C''s name-space which are not declared *private* by C'. The second approach requires that each object have a property `private` which, when set, makes the object invisible to external scripts.

Global Access Control List. A script in context C requests to establish a trust relationship via either a *open()* or *connect()* function call to its interpreter. The safe interpreter then has to decide if such a relationship is permissible. This decision depends on the *access control policy.* We make use of *access control lists (ACL).* ACLs are a well-known tools for implementing secure operating and file systems. In our model, an ACL is a set of contexts. For example, an ACL can be a set of URLs, where each URL represents a window's current content. We associate with each context C a single *access control list (ACL)*, which indicates the set of contexts which have the privilege to obtain access to N_C. The ACL is part of N_C and its value is loaded whenever C is loaded into a window. The ACL should only reside in N_C^r; otherwise, a script, possibly from a different context, can change the ACL.
Let us now define the *open()* and *connect()* functions:

- A script in C issues *open(C', w')*, where w' is a new (i.e., not currently open) window: If C is in C's ACL, then a new window w' is created and context C' is activated in w'; a trust relationship is established.
- A script in C issues *open(C', w')*, where w' is an open window with an active context C'' (possibly $C'' = C'$): If C is in C's ACL, then C''

is terminated and C' is activated; a trust relationship is established.

- A script in C issues *connect(C', w')*: If C' is already active in window w', a trust relationship is established; otherwise it is handled like an *open()* call.

Design Issue 6 for Secure JS: Global Access Control We introduce the property `document.ACL`, the document's access control list to be a member of N_C^r and N_C^i. In Secure JS, `window.open` implements the *open()* function described above. The required inputs are a *URL* and a *name*. The safe interpreter checks that `document.domain` in the context of the calling script is included in `document.ACL` of the context (defined via the *URL*) to be loaded. If this check succeeds, then the interpreter proceeds as follows: If *name* matches an open window w', then the current document (context) in w' is unloaded and the new context (*URL*) is loaded. If there is no match with an open window, then the interpreter opens a new window w' with *name* and loads the the new context (*URL*). If the access control check is positive, `window.open` returns a reference to the context in w'. The safe interpreter also checks if `document.domain` of w' is included in the calling script's `document.ACL`. If this check is positive, then a side effect of the this function call is that `w'.opener` provides w' with a reference the calling script's context. We also introduce a new method `window.connect` with the same arguments as `window.open`. This method differs from `window.open` only in the case where a window with name *name* and context *URL* already exists. In that case `connect` simply provides the two contexts with references to each other. Lastly, we introduce the new object property `private`. For Secure JS, each object in the object hierarchy has this additional property. If this property is set to `true`, the corresponding object is invisible to external scripts. After executing `victim = window.open("www.vendor.com", "spy")`, (and passing the ACL check) a script might try `snoop = victim.document.forms[0].elements[0].value`, assuming that in the `victim`-context, the first field in the first form asks the user for a credit-card number. If that field was specified to be *private*, then the safe interpreter would not execute the above statement. □

Local Access Control List. While the above approach implements a reasonable trust security policy (not unlike a UNIX file system enhanced with ACLs), it allows that "the right to exercise access carries with it the right to grant access" (as noted in the context of "unmodified" capability systems in, e.g., [B84, G89a, KL87]). For example, if a script in context C' is allowed to read (and hence copy) an object in context C, then every script which is in a context C'', such that C'' is in C''s ACL, can read this object. For more sensitive information, such transitivity of trust might not be suitable. For instance, bugs in the design of C' can lead to unintended values of the ACL of C', which then allows scripts of an adversarial context C'' to access data in C, undetectable to C. Many extensions to capability systems have been proposed to address this concern (see, e.g., [G89b, KH84, KL87]). In our environment, the following simple refinement of the above access list based control is sufficient: Besides a context-global ACL, each object in this context is associated with its own, local ACL (the same as the global ACL when the context is loaded). In the realm of secure operating systems, the data plus its data security attributes is called a *segment* (see, e.g., [KL87]). When an object gets copied (from one context to another), the segment remains intact, i.e., the safe interpreter copies the ACL as well. The *open* and *connect* routines work as described in the first solution; however, if a relationship is established, scripts in N_C get a reference only to those items in $N_{C'}$, for which C is in their local ACL. Hence, a script's context now plays the role of its *capability* (see [KL87]), which is checked against the ACL of the segment of the item made accessible. A local ACL (of a segment) is not part of the scripting language proper and consequently should be physically stored within the safe interpreter, inaccessible to any script. Segments copied from a different context might have a different ACL. With this security policy, bugs in context C' no longer allow access of third parties to context C.

A somewhat difficult aspect of the scheme described above, is to unequivocally define the semantics of "copying" an object, which guides when the object's ACL has to be updated. Assignments are straightforward. Many scripting language allow control flow constructs, such as *if ... then ... else* or *while ... do*. If a conservative policy is assumed, then assignments made in the scope of a control flow construct should be considered to be "copies" of values read in the control part. This is similar to, for example, the *tainting facility* of PERL (see [WCS96]).

Design Issue 7 for Secure JS: Local Access Control If Secure JS is used with the option of local access control, then every object in the JavaScript

object hierarchy has a property `ACL`, its local access control list. Once again, `ACL` property belongs in N_C^r. When a context is loaded and items in N_C^i are created, the value of their respective `ACL` property is set to the value of `window.document.ACL`. Items created by scripts (and hence in N_C^s) also have the same initial value of their `ACL`. Now consider the example, where a script in context C executed `window.open` which returned a reference *victim* to another window (context) C'. The script can now execute the following code: `creditCardNum = victim.document.forms[0].elements[0].value` assuming that the first element of the first form of context C' asks the user for a credit card number. A side effect of this code is the assignment `creditCardNum.ACL = victim.document.ACL`.
The script in C now executes `window.open` again to open another window with a third context C''. A script in C'' could potentially execute `snoop = window.opener.creditCardNum`. However, before this code is executed, the safe interpreter verifies that C'''s `document.domain` is included in `window.opener.creditCardNum.ACL`, permitting the operation only if the check succeeds. □

4.6 How the pieces fit together

In this section, we briefly show how the concepts of access control, independence of contexts, and trust management support data security and user privacy.
User Privacy: Almost all data on a user's machine which is not directly related to the current HTML document should be considered private to a user. *Access control* provides a padded cell for scripts, which assures that scripts cannot access the file system, process data and other unsafe resources. By maintaining a clear separation of data outside a name-space, read-only items, and writable items within the name-space, access control assures that scripts only have access to those parts of browser and window related data, which do not compromise a user's privacy while browsing. We have also described the *external interface* to a browser. Furthermore, *independence of contexts* assures that there are no "hidden channels" among different contexts. For example, if a writable item persisted across changes of context (as it is currently the case in JavaScript), it could be used as a user-invisible (albeit non-persistent) 'cookie' accessible to collaborating Web-sites.
Data Security: Data provided by a user in a context C of a window w (e.g., by filling out a form in the context's HTML document) is only available to scripts in a different context C', if C' is in C's ACL. Scripts in any other context however, are not able to access this data. Furthermore, setting the *private* property of an object guarantees that only scripts of the same context can access that object. This is assured by *trust management. Independence of contexts* assures that any context C' activated after C in w cannot access any data written in C. Furthermore, if C'' had a reference to a context C' that was loaded in w before C, this reference was set to null at the time C' was unloaded; assured by independence of contexts. Hence, only scripts in trusted contexts can access the user input data. Accesses to user data by external interfaces are guarded by the safe interpreter.

5 Security Analysis

In this section, we discuss scripting languages vulnerabilities exposed by different attacks, highlighting how they could have been prevented using a safe interpreter. We discuss other variants of this attack, as well as other problems with JavaScript in the Appendix.

5.1 Bell Labs Attack

Our security analysis of scripting languages on browsers from Netscape Corp. and Microsoft Corp. that are currently in use revealed that browser windows could be tricked into trusting attack scripts from rogue sites, thus allowing them to access their data (and hence the name-space of embedded scripts). See [CERT97, NN-Bug97, MSIE-Bug97]. Using this vulnerability, a rogue site could be set up to track all Web-related activity of visitors even after they had left the site, using a Trojan-horse attack. The tracking provided access to all data typed into forms (e.g., password fields), to cookies, and to visited URLs.
There were three requirements for the Trojan horse to successfully snoop on user activity. It needed to persist across multiple document fetches (script context changes), it needed to be able to access data from other loaded documents (across windows), and it needed to be able to transmit captured data to a desired location. A 'safe interpreter', properly implemented, would have safeguarded against all those attack components.

5.2 Initialization

Initially, the user loads an HTML document (context C) into a browser window w, which (unknown to the user) contains some adversarial script. This script first sets itself up to exist independently of the original window w, so it can persist across multiple document loads in w. To do this, it creates a new window (subsequently called the snooping window) w' using a *window.open()* call, and loads a different HTML document (context C') that contains the attack script into w'. Though a vigilant user could notice the presence of the new window, the creation of a new window is not a suspicious act in itself-many sites use multiple windows to display documents on the Web. In addition, control over size, placement and stacking of windows can be used effectively to reduce the chances of detection.

5.3 Data Extraction

The attack script in w' next gets a handle (object reference) to window w. In Netscape Navigator 2.*, this is done by setting a property of the snooping window w' when it was created by the adversarial script in window w. In all other browser versions, the *window.opener* property of the JavaScript object *window* gives the attack script in window w' the desired reference to w. We discuss more specifics of the vulnerability and the attack in the context of different browsers, highlighting the deficiencies that were exploited.

Netscape Navigator

Navigator 2.*, 3.*, and 4.* all use various means of trust management: (1) document address based access control - only a script from the HTTP server (domain name) that sent the HTML document is able to access data in a document, (2) a Perl-like 'tainting' model ([WCS96, F97, KK97]), and (3) a digital signature based scheme for access control [KK97], in which only signed scripts with the appropriate rights are able to access data in documents from other domains.

The attack script (in window w') bypassed security in all cases by using the window reference to insert arbitrary scripts into the observed window w's context in the form of JavaScript URLs (These are URLs of the form *javascript:<JavaScript code>*.) We discovered that code in these JavaScript URLs was not subject to access control and persisted across document reloads (context changes) in w. The inserted scripts periodically gathered all information available in the current context of the observed window w, and made it available to the snooping window w'. Data was stolen by directly writing into properties of the snooping window document from the observed window, in the case of Navigator 2.* and 3.*. In the case of Navigator 4.*, data was extracted by writing to and reading from a property of a mutually accessible object. Hence, the contexts of these two windows (w and w') were not disjoint!

Microsoft Internet Explorer

Microsoft Internet Explorer 3.* did not impose any security restrictions that needed to be bypassed. Starting with the window reference of the observed window w, attack scripts in the snooping window w' were free to walk the instance object hierarchy of the observed window w, and had direct access to all the information in any documents subsequently loaded into the observed window. The attack scripts periodically polled the current context C of the observed window w for its contents, copied the data, and then transmitted the captured data to a remote server.

We subsequently discovered that the same effect could be achieved using VBScript. This variant of the attack scripts used VBScript to set up a snooping window w', which was subsequently able to walk the instance hierarchy of the observed window w, and capture and transmit all the information.

The beta version of Microsoft IE 4.0 also has a security model centered around script address based access control. Once again, the exploit bypassed security by using JavaScript URLs to inject and execute arbitrary JavaScript code in the observed window, and to move captured data from the observed window to the snooping window.

5.4 Data Transmission

Once data was obtained, scripts in the snooping window w' were free to transmit it to a location of their choosing. They could transmit directly via HTTP by sending the data to a dynamically constructed URL. This could not be detected by the user. The snooping window could put the data into fields of a form and automatically submit the form. This method could be detected and circumvented by a vigilant user who had set up the browser to require interactive confirmation of all form submissions. Finally, code in the snooping window could communicate with an installed Java applet and/or ActiveX script to transmit the information to a desired location. This also could not be detected by the user.

5.5 Role of a Safe Interpreter in the Bell Labs Attack

The safe interpreter described in Section 4 would have protected against this style of attacks. When the attack was set up, context C in w, the observed window, and context C' in w', the snooping window, trusted each other, since they both originated on the attacker's Web-site. Proper trust management would have terminated the trust relationship when C was unloaded from w - trust doesn't persist across unloading of HTML documents (context changes.) The contexts would have been prevented from writing into each other. Enforcement of independence of contexts by the safe interpreter would subsequently have disallowed any accesses from C' to C (and vice versa) via the reference to C, rendering it useless for tracking. Finally, a regulated external interface could have been used to implement policies that alerted the user to potentially suspicious activity when the safe interpreter detected that an attempt was made to send data from a different site to the attacker's machine.

6 Conclusions

We have shown how taking steps towards a careful security design for scripting languages can help in preventing successful attacks on a user's security and privacy on the Web. It is not surprising that well known notions in access control, such as "ACL" and "capability" (see, e.g., [B84, G89a, G89b, KL87]) reappear, not only in our context, but also in ongoing research on how to design a more flexible Java (downloaded executable content) security architecture (see, e.g., [JRP96, G97, WBDF97]). It demonstrates that for a sound security model for downloadable, executable content on the Web, there are no shortcuts to a careful design based on notions and algorithms developed for secure operating systems.

References

[Anon] Anonymizer, http://www.anonymizer.com.

[B84] W. E. Boebert, *On the Inability of an Unmodified Capability Machine to Enforce the *-Property,* 7th DoD/NBS Computer Security Conference, 1984.

[B94] N. Borenstein, *Email with a mind of its own: The Safe-Tcl language for enabled mail,* IFIP Conference on Upper Layer Protocols, Architectures, and Applications, 1994.

[B97] D. Brumleve, *Tracker Privacy Bug,* Aleph2 Software, July 1997, http://www.aleph2.com/tracker/tracker.cgi.

[C97] K. Chiang, *Singapore Privacy Bug,* ITI, Singapore, July 1997, http://www.iti.gov.sg/iti_people/iti_staff/kcchiang/bug/.

[CERT97] CERT* Advisory CA-97.20, *JavaScript Vulnerability,* CERT Coordination Center, July 1997, ftp://info.cert.org/pub/cert_advisories/CA-97.20.javascript.

[F97] D. Flanagan, *JavaScript: The Definitive Guide,* O'Reilly and Associates, January 1997.

[G89a] L. Gong, *On Security in Capability-Based Systems,* ACM Operating Systems Review, 1989.

[G89b] L. Gong, *A Secure Identity-Based Capability System,* IEEE Symposium on Security and Privacy, 1989.

[G97] L. Gong, *New Security Architectural Directions for Java,* IEEE COMPCON, 1997.

[JRP96] T. Jaeger, A. Rubin, A. Prakash, *Building systems that flexibly control downloaded executable content,* 6th USENIX Security Symposium, 1996.

[KH84] P. A. Karger, A. J. Herbert, *An augmented capability architecture to support lattice security and traceability of access,* Proc. 1984 IEEE Symp. on Security and Privacy.

[KK97] P. Kent, J. Kent, *Official Netscape JavaScript 1.2 Book,* Netscape Press & Ventana.

[KL87] R. Y. Kain and C. E. Landwehr, *On Access Checking in Capability-Based Systems.* IEEE Transactions on Software Engineering, Vol **SE-13**, No **2**, February 1987.

[L96] J. R. LoVerso, *JavaScript Security Flaws,* OSF, http://www.osf.org/ loverso/javascript/.

[L97] P. Lomax, *Learning VBScript,* O'Reilly and Associates, July 1997.

[LPWA] Lucent Personalized Web Assistant, http://lpwa.com:8000.

[MF97] G. McGraw and E. Felten, *Java Security: Hostile Applets, Holes, and Antidotes.* Wiley Computer Publishing, 1997.

[MSIE-Bug97] *Bell Labs Privacy Problem*, Microsoft Corp., July 1997, http://www.microsoft.com/ie/security/bell.htm.

[NN-Bug97] *Bell Labs Privacy Bug*, Netscape Corp., July 1997, http://www.netscape.com/assist/security/resources/bell_labs_privacy.html.

[OLW96] J. Ousterhout, J. Levy, B. Welch, *The Safe-Tcl Security Model*, Manuscript.

[S97] A. dos Santos, Santa Barbara Privacy Bug, *UCSB Secure Internet Programming Team*, August 1997, http://www.cs.ucsb.edu/ andre/Attack.html.

[WBDF97] D. Wallach, D. Balfanz, D. Dean, E. Felten, *Extensible Security Architectures for Java.* Princeton CS, TR-546-97, April 1997

[WCS96] L. Wall, T. Christiansen, R. Schwartz, *Programming Perl.* O'Reilly & Associates, Inc., September 1996.

Appendix: Other Attacks

In this section we describe some of the other browser scripting language based attacks that were reported.

The Tracker attack

An attack that behaved similar to the Bell Labs attack, and worked specifically in Netscape Navigator 3.0 was devised independently, see [B97], and was equally effective at monitoring user activity on the Web. It worked even for Navigator 3.02, which incorporated a patch for the Bell Labs attack! The *onUnload()* event handler for document objects is a piece of JavaScript code that is executed when the browser unloads the current HTML document before replacing it with a new document. This attack worked by using attack scripts in an attack window w' (context) to dynamically insert a new *onUnload()* handler in the observed browser window w. This newly inserted code would be activated when the current document C was unloaded, and would walk through the object instance hierarchy and surreptitiously transmit all accessible information to a remote server.

A safe interpreter would not allow scripts of an untrusted context C' to insert code into context C, thus disallowing subversion of the *onUnload()* event handler.

Singapore Attack

Another similar attack was devised independently (see [C97]) and worked specifically in Netscape Navigator 4.*. It was even more insidious. This Trojan horse did not use a separate window to persist across multiple document fetches, and was thus harder to detect. The Java VM in browsers allows applet threads to continue to run even after the document that started the applet is unloaded. (A discussion of Java and its security is outside the scope of this document, see, e.g., [MF97]). LiveConnect [F97] is a 'glue' technology from Netscape Corp. that is designed to allow Java applets, browser plugins and JavaScript scripts to communicate in Netscape's browsers. It is manifested as an API that programmers can use to implement desired interaction between these different entities. An applet can communicate with scripts by using LiveConnect Java classes to get a window handle (reference to a JavaScript context). The attack applet obtained a window handle of its own window w, which remained valid even after the HTML document C that initially loaded the applet was unloaded, and another HTML document C' was loaded into the window. The applet could then, via the window handle, walk through the JavaScript object instance hierarchy of the new document, access any contained information, and transmit it to a remote server.

A safe interpreter would invalidate the reference to a context C (including references given to external interfaces) as soon as C was unloaded and thus would prevent the Singapore attack. This attack however, is a reminder that interaction between different entities with different security policies is not well understood and always a potential threat.

This point was further substantiated by the Santa Barbara attack (see [S97]), which produced similar results in Netscape Communicator 4.02. It used a Java applet to access and steal data in HTML documents in an observed window W by tricking it into executing attack JavaScript code.

Other Scripting Attacks

From the start, JavaScript implementations inadvertently allowed malicious scripts to attack a user in different ways [L96].

Early implementations of JavaScript stored the user's browsing history into the *history* object, and allowed scripts read-access to the whole object. Malicious scripts were free to transmit this information to a server, allowing operators to trivially build dossiers on Web users. Another problem, discovered early on, was the complete absence of access control! This allowed malicious scripts to actively (and easily) track a user's browsing history by monitoring another browser window. A variant of this method allowed scripts to browse directories in a user's machine by starting with a document obtained via a URL using the *file:* protocol - the script could walk through the instance hierarchy of a dynamically generated document that represented the directory and its contents from the local file system, and transmit this information to a remote server. In another instance, the interpreter let scripts from one document (context) stay resident even after the document was unloaded, opening the door to variants of the above attacks.

Browsers began to support the *mailto:* protocol for submitting forms on the Web, which used the E-mail infrastructure for transport, instead of HTTP. Automatic submission of forms by scripts, using the *mailto:* protocol (and hence, the browser as an external interface) allowed Web-site operators to surreptitiously capture the email address of users. File Upload elements in HTML-forms provided a mechanism for Web users to transmit (or upload) files from the local file system to a remote Web server. Malicious scripts, however, were able to specify the absolute path of an arbitrary file, and cause the browser (again as an external interface) to automatically submit the form, transmitting the contents of that file to the remote server.

The fix for these problems typically involved hobbling the responsible feature - scripts were no longer allowed to access the list of URLs in the *history* object; a script-address based access control policy was developed - JavaScript scripts could read properties of a document only if the scripts were loaded from the same server (domain name) as the document; automatic submission of forms using the *mailto:* protocol was disallowed; and File Upload elements in forms could have their value set only interactively, not by a Web-site operator.

An insidious variant of the File Upload attack was later discovered in Netscape browsers. It exploited an obscure bug in the implementation of JavaScript that allowed an attack script to assign an arbitrary file name to a File Upload element of a form. (An implementation of this attack violated a name-space rule of the safe interpreter by adding elements to N_C^i, the interpreter-created object set, after the context C was already activated!) The specified file would be transmitted to the Web server when the form was submitted, either automatically by JavaScript or as a result of a user-initiated submission.

The recently reported French Privacy Bug in Netscape Communicator 4.02 allowed a malicious script to read and transmit a user's local browser preferences file if the script could successfully guess it's location on the local file-system. The Freiburg Issue affected Microsoft Internet Explorer 4.0, and allowed a script to read and transmit local text, HTML or image files if the JScript or VBScript script could successfully guess their location on the file-system.

Though most of these flaws were fixed as scripting implementations evolved, a formal specification of a safe interpreter or safe interaction with a browser vis-a-vis scripting has not been developed and publicly released.

Access control in the safe interpreter formally specifies what a script is allowed to access in the user's environment - hence policies on access to *history* information, user information, files etc. can easily be implemented. Regulated external interfaces can be used to implement policies that safeguard against surreptitious capture of information via HTTP, FTP and SMTP, or from the file-system. Finally, the formalism of a safe interpreter enables one to reason about browser scripting at the level of semantics, rather than mechanics of different operations.

Finite-State Analysis of SSL 3.0

John C. Mitchell Vitaly Shmatikov Ulrich Stern

Computer Science Department
Stanford University
Stanford, CA 94305

{jcm, shmat, uli}@cs.stanford.edu

Abstract

The Secure Sockets Layer (SSL) protocol is analyzed using a finite-state enumeration tool called Murφ. The analysis is presented using a sequence of incremental approximations to the SSL 3.0 handshake protocol. Each simplified protocol is "model-checked" using Murφ, with the next protocol in the sequence obtained by correcting errors that Murφ finds automatically. This process identifies the main shortcomings in SSL 2.0 that led to the design of SSL 3.0, as well as a few anomalies in the protocol that is used to resume a session in SSL 3.0. In addition to some insight into SSL, this study demonstrates the feasibility of using formal methods to analyze commercial protocols.

1 Introduction

In previous work [9], a general-purpose finite-state analysis tool has been successfully applied to the verification of small security protocols such as the Needham-Schroeder public key protocol, Kerberos, and the TMN cellular telephone protocol. The tool, Murφ [3, 10], was designed for hardware verification and related analysis. In an effort to understand the difficulties involved in analyzing larger and more complex protocols, we use Murφ to analyze the SSL 3.0 handshake protocol. This protocol is important, since it is the de facto standard for secure Internet communication, and a challenge, since it has more steps and greater complexity than the other security protocols analyzed using automated finite-state exploration. In addition to demonstrating that finite-state analysis is feasible for protocols of this complexity, our study also points to several anomalies in SSL 3.0. However, we have not demonstrated the possibility of compromising sensitive data in any implementation of the protocol.

In the process of analyzing SSL 3.0, we have developed a "rational reconstruction" of the protocol. More specifically, after initially attempting to familiarize ourselves with the handshake protocol, we found that we could not easily identify the purpose of each part of certain messages. Therefore, we set out to use our analysis tool to identify, for each message field, an attack that could arise if that field were omitted from the protocol. Arranging the simplified protocols in the order of increasing complexity, we obtain an incremental presentation of SSL. Beginning with a simple, intuitive, and insecure exchange of the required data, we progressively introduce signatures, hashed data, and additional messages, culminating in a close approximation of the actual SSL 3.0 handshake protocol.

In addition to allowing us to understand the protocol more fully in a relatively short period of time, this incremental reconstruction also provides some evidence for the "completeness" of our analysis. Specifically, Murφ exhaustively tests all possible interleavings of protocol and intruder actions, making sure that a set of correctness conditions is satisfied in all cases. It is easy for such analysis to be "incomplete" by not checking all of the correctness conditions intended by the protocol designers or users. In developing our incremental reconstruction of SSL 3.0, we were forced to confirm the importance

This work was supported in part by the Defense Advanced Research Projects Agency through NASA contract NAG-2-891, Office of Naval Research grant N00014-97-1-0505, Multidisciplinary University Research Initiative "Semantic Consistency in Information Exchange", National Science Foundation grant CCR-9629754, and the Hertz Foundation. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, NASA, ONR, NSF or the US Government.

of each part of each message. In addition, since no formal or high-level description of SSL 3.0 was available, we believe that the description of SSL 3.0 that we extracted from the Internet Draft [6] may be of interest.

Our analysis covers both the standard handshake protocol used to initiate a secure session and the shorter protocol used to resume a session [6, Section 5.5]. Murφ analysis uncovered a weak form of version rollback attack (see Section 4.9.2) that can cause a version 3.0 client and a version 3.0 server to commit to SSL 2.0 when the protocol is resumed. Another attack on the resumption protocol (described in Sections 4.8 and 4.9.1) is possible in SSL implementations that strictly follow the Internet draft [6] and allow the participants to send application data without waiting for an acknowledgment of their *Finished* messages. Finally, an attack on cryptographic preferences (see Section 4.6) succeeds if the participants support weak encryption algorithms which can be broken in real time. Apart from these three anomalies, we were not able to uncover any errors in our final protocol. Since SSL 3.0 was designed to be backward-compatible, we also implemented and checked a full model for SSL 2.0 as part of the SSL 3.0 project. In the process, Murφ uncovered the major problems with SSL 2.0 that motivated the design of SSL 3.0.

Our Murφ analysis of SSL is based on the assumption that cryptographic functions cannot be broken. For this and other reasons (discussed below), we cannot claim that we found all attacks on SSL. But our analysis has been efficient in helping discover an important class of attacks.

The two prior analyses of SSL 3.0 that we are aware of are an informal assessment carried out by Wagner and Schneier [14] and a formal analysis by Dietrich using a form of belief logic [2]. (We read the Wagner and Schneier study before carrying out our analysis, but did not become aware of the Dietrich study until after we had completed the bulk of our work.) Wagner and Schneier comment on the possibility of anomalies associated with resumption, which led us to concentrate our later efforts on this area. It is not clear to us at the time of this writing whether we found any resumption anomalies that were not known to these investigators. However, in email comments resulting from circulation of an earlier document [13], we learned that while our second anomaly was not noticed by Wagner and Schneier, it was later reported to them by Michael Wiener. Neither anomaly seems to have turned up in the logic-based study of [2].

A preliminary report on this study was presented in a panel at the September 1997 DIMACS Workshop on Design and Formal Verification of Security Protocols and appears on the web site (`http://dimacs.rutgers.edu/Workshops/Security/`) and CD ROM associated with this workshop. Our study of resumption was carried out after our workshop submission and is not described in the workshop document.

2 Outline of the methodology

Our general methodology for modeling security protocols in Murφ is described in our previous paper [9], and will be only outlined in this section. The basic approach is similar to the CSP approach to model checking of cryptographic protocols described in [8, 11].

2.1 The Murφ verification system

Murφ [3] is a protocol or, more generally, finite-state machine verification tool. It has been successfully applied to several industrial protocols, especially in the domains of multiprocessor cache coherence protocols and multiprocessor memory models [4, 12, 15]. The purpose of finite-state analysis, commonly called "model checking," is to exhaustively search all execution sequences. While this process often reveals errors, failure to find errors does not imply that the protocol is completely correct, because the Murφ model may simplify certain details and is inherently limited to configurations involving a small number of, say, clients and servers.

To use Murφ for verification, one has to model the protocol in the Murφ language and augment this model with a specification of the desired properties. The Murφ system automatically checks, by explicit state enumeration, if all reachable states of the model satisfy the given specification. For the state enumeration, either breadth-first or depth-first search can be selected. Reached states are stored in a hash table to avoid redundant work when a state is revisited. The memory available for this hash table typically determines the largest tractable problem.

The Murφ language is a simple high-level language for describing nondeterministic finite-state machines. Many features of the language are familiar from conventional programming languages. The main features not found in a "typical" high-level language are described in the following paragraphs.

The *state* of the model consists of the values of all global variables. In a *startstate* statement, initial values are assigned to global variables. The transition from one state to another is performed by *rules.*

Each rule has a Boolean condition and an action, which is a program segment that is executed atomically. The action may be executed if the condition is true (i.e., the rule is enabled) and typically changes global variables, yielding a new state. Most Murφ models are nondeterministic since states typically allow execution of more than one rule. For example, in the model of the SSL protocol, the intruder (which is part of the model) usually has the nondeterministic choice of several messages to replay.

Murφ has no explicit notion of *processes.* Nevertheless a process can be implicitly modeled by a set of related rules. The *parallel composition* of two processes in Murφ is simply done by using the union of the rules of the two processes. Each process can take any number of steps (actions) between the steps of the other. The resulting computational model is that of *asynchronous, interleaving* concurrency. Parallel processes communicate via shared variables; there are no special language constructs for communication.

The Murφ language supports *scalable* models. In a scalable model, one is able to change the size of the model by simply changing constant declarations. When developing protocols, one typically starts with a small protocol configuration. Once this configuration is correct, one gradually increases the protocol size to the largest value that still allows verification to complete. In many cases, an error in the general (possibly infinite state) protocol will also show up in a down-scaled (finite state) version of the protocol. Murφ can only guarantee correctness of the down-scaled version of the protocol, but not correctness of the general protocol. For example, in the model of the SSL protocol, the numbers of clients and servers are scalable and defined by constants.

The desired properties of a protocol can be specified in Murφ by invariants, which are Boolean conditions that have to be true in every reachable state. If a state is reached in which some invariant is violated, Murφ prints an error trace – a sequence of states from the start state to the state exhibiting the problem.

2.2 The methodology

In outline, we have analyzed protocols using the following sequence of steps:

1. *Formulate the protocol.* This generally involves simplifying the protocol by identifying the key steps and primitives. The Murφ formulation of a protocol, however, is more detailed than the high-level descriptions often seen in the literature, since one has to decide exactly which messages will be accepted by each participant in the protocol. Since Murφ communication is based on shared variables, it is also necessary to define an explicit message format, as a Murφ type.

2. *Add an adversary to the system.* We generally assume that the adversary (or intruder) can masquerade as an honest participant in the system, capable of initiating communication with a truly honest participant, for example. We also assume that the network is under control of the adversary and allow the adversary the following actions:

   - overhear every message, remember all parts of each message, and decrypt ciphertext when it has the key,
   - intercept (delete) messages,
   - generate messages using any combination of initial knowledge about the system and parts of overheard messages.

   Although it is simplest to formulate an adversary that nondeterministically chooses between all possible actions at every step of the protocol, it is more efficient to reduce the choices to those that actually have a chance of affecting other participants.

3. *State the desired correctness condition.* A typical correctness criterion includes, e.g., that no secret information can be learned by the intruder. More details about the correctness criterion used for our SSL model are given in Section 3.

4. *Run the protocol* for some specific choice of system size parameters. Speaking very loosely, systems with 4 or 5 participants (including the adversary) and 5 to 7 intended steps in the original protocol (without adversary) are easily analyzed in minutes of computation time using a modest workstation. Doubling or tripling these numbers, however, may cause the system to run for many hours, or terminate inconclusively by exceeding available memory.

5. *Experiment with alternate formulations and repeat.* This is discussed in detail in Section 4.

2.3 The intruder model

The intruder model described above is limited in its capabilities and does not have all the power that a real-life intruder may have. In the following, we discuss examples of these limitations.

No cryptanalysis. Our intruder ignores both computational and number-theoretic properties of cryptographic functions. As a result, it cannot perform any cryptanalysis whatsoever. If it has the proper key, it can read an encrypted message (or forge a signature). Otherwise, the only action it can perform is to store the message for a later replay. We do not model any cryptographic attacks such as brute-force key search (with a related notion of computational time required to attack the encryption) or attacks relying on the mathematical properties of cryptographic functions.

No probabilities. Murφ has no notion of probability. Therefore, we do not model "propagation" of attack probabilities through our finite-state system (e.g, how the probabilities of breaking the encryption, forging the signature, etc. accumulate as the protocol progresses). We also ignore, e.g., that the intruder may learn some probabilistic information about the participants' keys by observing multiple runs of the protocol.

No partial information. Keys, nonces, etc. are treated as atomic entities in our model. Our intruder cannot break such data into separate bits. It also cannot perform an attack that results in the partial recovery of a secret (e.g., half of the secret bits).

In spite of the above limitations, we believe that Murφ is a useful tool for analyzing security protocols. It considers the protocol at a high level and helps discover a certain class of bugs that do not involve attacks on cryptographic functions employed in the protocol. For example, Murφ is useful for discovering "authentication" bugs, where the assumptions about key ownership, source of messages, etc. are implicit in the protocol but never verified as part of the message exchange. Also, Murφ models can successfully discover attacks on plaintext information (such as version rollback attacks in SSL) and implicit assumptions about message sequence in the protocol (such as unacknowledged receipt of *Finished* messages in SSL). Other examples of errors discovered by finite-state analysis appear in [8, 9, 11] and in references cited in these papers.

3 The SSL 3.0 handshake protocol

The primary goal of the SSL 3.0 handshake protocol is to establish secret keys that "provide privacy and reliability between two communicating applications" [6]. Henceforth, we call the communicating applications the client (C) and the server (S). The basic approach taken by SSL is to have C generate a fresh random number (the *secret* or *shared secret*) and deliver it to S in a secure manner. The secret is then used to compute a so-called *master secret* (or *negotiated cipher*), from which, in turn, the keys that protect and authenticate subsequent communication between C and S are computed. While the SSL *handshake protocol* governs the secret key computation, the SSL *record layer protocol* governs the subsequent secure communication between C and S.

As part of the handshake protocol, C and S exchange their respective *cryptographic preferences*, which are used to select a mutually acceptable set of algorithms for encrypting and signing handshake messages. In our analysis, we assume for simplicity that RSA is used for both encryption and signatures, and cryptographic preferences only indicate the desired lengths of keys. In addition, SSL 3.0 is designed to be backward-compatible so that a 3.0 server can communicate with a 2.0 client and vice versa. Therefore, the parties also exchange their respective version numbers.

The basic handshake protocol consists of three messages. With the *ClientHello* message, the client starts the protocol and transmits its version number and cryptographic preferences to the server. The server replies with the *ServerHello* message, also transmitting its version number and cryptographic preferences. Upon receipt of this message, the client generates the shared secret and sends it securely to the server in the *secret exchange message.*

Since we were not aware of any formal definition of SSL 3.0, we based our model of the handshake protocol on the Internet draft [6]. The draft does not include a precise list of requirements that must be satisfied by the communication channel created after the handshake protocol completes. Based on our interpretation of the informal discussion in Sections 1 and 5.5 of the Internet draft, we believe that the resulting channel can be considered "secure" if and only if the following properties hold:

- Let $Secret_C$ be the number that C considers the shared secret, and $Secret_S$ the number that S considers the shared secret. Then $Secret_C$ and $Secret_S$ must be identical.

- The secret shared between C and S is not in intruder's database of known message components.

- The parties agree on each other's identity and protocol completion status. Suppose that the last message of the handshake protocol is from

S to C. Then C should reach the state in which it is ready to start communicating with S using the negotiated cipher ($Done_C$) only if S is already in the state in which it is ready to start communicating with C using the negotiated cipher ($Done_S$). Conversely, S should reach the state $Done_S$ only if C is in the state in which it is waiting for the last message of the handshake protocol.

- The cryptographic algorithms selected by the parties for encryption and authentication of handshake messages are the strongest ones that are supported by both C and S. We model this by requiring that the cryptosuite stored by S as C's cryptographic preferences is identical to the one actually sent by C, and vice versa.

- The parties have a consistent opinion about each other's version, i.e., it is never the case that an SSL 3.0 client and a 3.0 server are communicating using the SSL 2.0 protocol.

We propose that any violation of the foregoing invariants that goes undetected by the legitimate participants constitutes a successful attack on the protocol.

SSL 3.0 supports *protocol resumption.* In the initial run of the protocol, C and S establish a shared secret by going through the full protocol and computing secret keys that protect subsequent communication. SSL 3.0 allows the parties to resume their connection at a later time without repeating the full protocol. If the *ClientHello* message sent by C to S includes the identifier of an SSL session that is still active according to S's internal state, S assumes that C wants to resume a previous session. No new secret is exchanged in this case, but the master secret and the keys derived from it are recomputed using new nonces. (See Section 4.8 for an explanation of how nonces are used in the protocol to prevent replay attacks, and Appendix B to see how the master secret is computed from the nonces and shared secret.) Our Murφ model supports protocol resumption.

Finally, it should be noted that whenever one of the parties detects an inconsistency in the messages it receives, or any of the protocol steps fails in transmission, the protocol is aborted and the parties revert to their initial state. This implies that SSL is susceptible by design to some forms of "denial of service" attacks: an intruder can simply send an arbitrary message to a client or server engaged in the handshake protocol, forcing protocol failure.

4 Modeling SSL 3.0

We start our incremental analysis with the simplest and most intuitive version of the protocol and give an attack found by Murφ. We then add a little piece of SSL 3.0 that foils the attack, and let Murφ discover an attack on the augmented protocol. We continue this iterative process until no more attacks can be found. The final protocol closely resembles SSL 3.0, with some simplifications that result from our assumption of perfect cryptography (see below).

4.1 Notation

The following notation will be used throughout the paper.

Ver_i	SSL version number of party i
$Suite_i$	Cryptographic preferences of party i
N_i	Random nonce generated by party i
$Secret_i$	Random secret generated by party i
K_i^+	Public encryption key of party i
V_i	Public verification key of party i
$\text{sign}_i\{\ldots\}$	Signed by party i
$\{\ldots\}_{K_i^+}$	Encrypted by public key K_i^+
Messages	All messages up to this point
$\langle I \rangle$	Message is intercepted by the intruder

4.2 Assumptions about cryptography

In general, our model assumes perfect cryptography. The following list explains what this assumption implies for all cryptographic functions used in SSL.

Opaque encryption. Encryption is assumed to be opaque. If a message has the form $\{x\}_{K_i^+}$, only party i can learn x. (This is only true iff the private key K_i^- is not available to any party except i. This is a safe assumption, given that no participants in the SSL handshake protocol are ever required to send their private key over the network.) The intruder may, however, store the entire encrypted message and replay it later without learning x. The structure of the encrypted message is inaccessible to the intruder, i.e., it cannot split the encrypted message into parts and insert them into other encrypted messages.

Unforgeable signatures. Signatures are assumed to be unforgeable. Messages of the form $\text{sign}_i\{x\}$ can only be generated by the party i. Anyone who possesses i's verification key V_i

is able to verify that the message was indeed signed by i. We assume that signatures do not encrypt. Therefore, x can be learned by anyone.

Hashes. Hashes are assumed to be preimage resistant and 2nd-preimage resistant: given a message of the form $\text{Hash}\{x\}$, it is not computationally feasible to discover x, nor find any x' such that $\text{Hash}\{x'\} = \text{Hash}\{x\}$. It is therefore assumed that a participant can determine whether $x = x'$ by comparing $\text{Hash}\{x\}$ to $\text{Hash}\{x'\}$.

Trusted certificate authority. There exists a trusted certificate authority (CA). All parties are assumed to possess CA's verification key V_{CA}, and are thus able to verify messages signed by CA. Every party i is assumed to possess CA-signed certificates for its own public keys: $\text{sign}_{CA}\{i, K_i^+\}$ (certifying that public encryption key K_i^+ indeed belongs to i) and $\text{sign}_{CA}\{i, V_i\}$ (certifying that public verification key V_i indeed belongs to i).

4.3 Protocol A

Basic protocol (A)

The first step of the basic protocol consists of C sending information about its identity, SSL version number, and cryptographic preferences (aka *cryptosuite*) to S. Upon receipt of C's *Hello* message, S sends back its version, cryptosuite (S selects one set of algorithms from the preference list submitted by C), and its public encryption key. C then generates a random secret and sends it to S, encrypted by S's public key.

Notice that the first *Hello* message (that from C to S) contains the identity of C. There is no way for S to know who initiated the protocol unless this information is contained in the message itself (perhaps implicitly in the network packet header).

$C \rightarrow S$	$C,\ Ver_C,\ Suite_C$
$S \rightarrow C$	$Ver_S,\ Suite_S,\ K_S^+$
$C \rightarrow S$	$\{Secret_C\}_{K_S^+}$

⟨Change to negotiated cipher⟩

Attack on A

Protocol A does not explicitly (and securely) associate the server's name with its public encryption key. This allows the intruder to insert its own key into the server's *Hello* message. The client then encrypts the generated secret with the intruder's key, enabling the intruder to read the message and learn the secret.

$C \rightarrow S$	$C,\ Ver_C,\ Suite_C$
$S \rightarrow C\langle I\rangle$	$Ver_S,\ Suite_S,\ K_S^+$
$I \rightarrow C$	$Ver_S,\ Suite_S,\ \underline{K_I^+}$
$C \rightarrow S\langle I\rangle$	$\{Secret_C\}_{K_I^+}$
$I \rightarrow S$	$\{Secret_C\}_{K_S^+}$

⟨Change to negotiated cipher⟩

4.4 Protocol B

A + server authentication

To fix the bug in Protocol A, we add verification of the public key. The server now sends its public key K_S^+ in a certificate signed by the certificate authority. As described before, the certificate has the following form: $\text{sign}_{CA}\{S, K_S^+\}$.

We assume that signatures are unforgeable. Therefore, the intruder will not be able to generate $\text{sign}_{CA}\{S, K_I^+\}$. The intruder may send the certificate for its own public key $\text{sign}_{CA}\{I, K_I^+\}$, but the client will reject it since it expects S's name in the certificate. Finally, the intruder may generate $\text{sign}_I\{S, K_I^+\}$, but the client expects a message signed by CA, and will try to verify it using CA's verification key. Verification will fail since the message is not signed by CA, and the client will abort the protocol. Notice that SSL's usage of certificates to verify the server's public key depends on the trusted certificate authority assumption (see Section 4.2 above).

$C \rightarrow S$	$C,\ Ver_C,\ Suite_C$
$S \rightarrow C$	$Ver_S,\ Suite_S,\ \text{sign}_{CA}\{S, K_S^+\}$
$C \rightarrow S$	$\{Secret_C\}_{K_S^+}$

⟨Change to negotiated cipher⟩

Attack on B

Protocol B includes no verification of the client's identity. This allows the intruder to impersonate the client by generating protocol messages and pretending they originate from C. In particular, the intruder is able to send its own secret to the server, which the latter will use to compute the master secret and the derived keys.

$\underline{I} \to S$	$C,\ Ver_C,\ Suite_C$
$S \to C\langle I\rangle$	$Ver_S,\ Suite_S,\ \text{sign}_{CA}\{S, K_S^+\}$
$I \to S$	$\{\underline{Secret_I}\}_{K_S^+}$

⟨Change to negotiated cipher⟩

4.5 Protocol C

B + client authentication

To fix the bug in Protocol B, the server has to verify that the secret it received was indeed generated by the party whose identity was specified in the first *Hello* message. For this purpose, SSL employs client signatures.

The client sends to the server its verification key in the CA-signed certificate $\text{sign}_{CA}\{C, V_C\}$. In addition, immediately after sending its secret encrypted with the server's public key, the client signs the hash of the secret $\text{sign}_C\{\text{Hash}\,(Secret_C)\}$ and sends it to the server. Hashing the secret is necessary so that the intruder will not be able to learn the secret even if it intercepts the message. Since the server can learn the secret by decrypting the client key exchange message, it is able to compute the hash of the secret and compare it with the one sent by the client.

Notice that the server can be assured that V_C is indeed C's verification key since the intruder cannot insert its own key in the CA-signed certificate $\text{sign}_{CA}\{C, V_C\}$ assuming that signatures are unforgeable. Therefore, the server will always use V_C to verify messages ostensibly signed by the client, and all messages of the form $\text{sign}_I\{\ldots\}$ will be rejected. Even if the intruder were able to generate the message $\text{sign}_C\{\text{Hash}\,(Secret_I)\}$, the attack will be detected when the server computes $\text{Hash}\,(Secret_C)$ and discovers that it is different from $\text{Hash}\,(Secret_I)$.

Instead of signing the hashed secret, the client can sign the secret directly and send it to the server encrypted by the server's public key. The SSL definition, however, does not include encryption in this step [6, Section 5.6.8]. We used hashing instead of encryption as well since we intend our incremental reconstruction of SSL to follow the definition as closely as possible. One of the anonymous reviewers suggested that hashing is used instead of encryption so that the encrypted part of the message (i.e., a secret as opposed to a signed secret) fits within the modulus size of the server's encryption function.

$C \to S$	$C,\ Ver_C,\ Suite_C$
$S \to C$	$Ver_S,\ Suite_S,\ \text{sign}_{CA}\{S, K_S^+\}$
$C \to S$	$\text{sign}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+},$ $\text{sign}_C\{\text{Hash}\,(Secret_C)\}$

⟨Change to negotiated cipher⟩

Attack on C

Even though the intruder can modify neither keys nor shared secret in Protocol C, it is able to attack the plaintext information transmitted in the *Hello* messages. This includes the parties' version numbers and cryptographic preferences.

By modifying version numbers, the intruder can convince an SSL 3.0 client that it is communicating with a 2.0 server, and a 3.0 server that it is communicating with a 2.0 client. This will cause the parties to communicate using SSL 2.0, giving the intruder an opportunity to exploit any of the known weaknesses of SSL 2.0.

By modifying the parties' cryptographic preferences, the intruder can force them into selecting a weaker encryption and/or signing algorithm than they normally would. This may make it easier for the intruder to decrypt the client's secret exchange message, or to forge the client's signature.

$C \to S\langle I\rangle$	$C,\ Ver_C,\ Suite_C$
$I \to S$	$C,\ \underline{Ver_I},\ \underline{Suite_I}$
$S \to C\langle I\rangle$	$Ver_I,\ Suite_S,\ \text{sign}_{CA}\{S, K_S^+\}$
$I \to C$	$Ver_I,\ \underline{Suite_I},\ \text{sign}_{CA}\{S, K_S^+\}$
$C \to S$	$\text{sign}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+},$ $\text{sign}_C\{\text{Hash}\,(Secret_C)\}$

⟨Change to negotiated cipher⟩

4.6 Protocol D

C + post-handshake verification of plaintext

The parties can prevent attacks on plaintext by repeating the exchange of versions and cryptographic preferences once the handshake protocol is complete; the additional messages will be called *verification messages.* Since the intruder cannot learn the shared secret, it cannot compute the master secret and the derived keys and thus cannot interfere with the parties' communication after they switch to the negotiated cipher.

Suppose the intruder altered the cryptographic preferences in the client's *Hello* message. When the client sends its version and cryptosuite to the server under the just negotiated encryption, the intruder cannot change them. The server will detect the discrepancy and abort the protocol. This is also true for the server's version and cryptosuite.

$C \rightarrow S$	$C,\ Ver_C,\ Suite_C$
$S \rightarrow C$	$Ver_S,\ Suite_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$C \rightarrow S$	$\mathrm{sign}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+},$ $\mathrm{sign}_C\{\mathrm{Hash}\,(Secret_C)\}$
⟨Change to negotiated cipher⟩	
$S \rightarrow C$	$\{\mathrm{Hash}\,(Ver_C, Suite_C, Ver_S,$ $Suite_S)\}_{\mathrm{Master}(Secret_C)}$
$C \rightarrow S$	$\{\mathrm{Hash}\,(Ver_C, Suite_C, Ver_S,$ $Suite_S)\}_{\mathrm{Master}(Secret_C)}$

The above protocol is secure against attacks on version numbers and cryptographic preferences except in the following circumstances:

1. If an attack on version number in the first *Hello* message causes the parties to switch to a different protocol such as SSL 2.0, they will not exchange verification messages and the attack will not be detected. See Section 4.9.2 for further discussion of anomalies related to the version rollback attack.

2. By changing cryptosuites in the *Hello* messages, the intruder may force the parties to use a very weak public-key encryption algorithm that can be broken in real time (i.e., while the current run of the handshake protocol is in progress). If the intruder can break the encrypted message containing the client's secret, it can compute the master secret and the derived keys and will thus be able to forge post-handshake verification messages. The only defense against this kind of attack is to prohibit SSL implementations from using weak cryptographic algorithms in the handshake protocol even if hello messages from the protocol counterparty indicate preference for such algorithms.

Attack on D

In Protocol D, the parties verify only plaintext information after the handshake negotiation is complete. Since the intruder cannot forge signatures, invert hash functions, or break encryption without the correct private key, it can neither learn the client's secret, nor substitute its own. It may appear that D provides complete security for the communication channel between C and S. However, Murφ discovered an attack on client's identity that succeeds even if all cryptographic algorithms are perfect.

Intruder I intercepts C's hello message to server S, and initiates the handshake protocol with S *under its own name.* All messages sent by S are retransmitted to C, while most of C's messages, including the post-handshake verification messages, are re-transmitted to S. (See the protocol run below for details. Re-transmission of C's verification message is required to change the sender identifier, which is not shown explicitly below.) As a result, both C and S will complete the handshake protocol successfully, but C will be convinced that it is talking to S, while S will be convinced that it is talking to I.

Notice that I does not have access to the secret shared between C and S. Therefore, it will not be able to generate or decrypt encrypted messages after the protocol is complete, and will only be able to re-transmit C's messages. However, the server will believe that the messages are coming from I, whereas in fact they were sent by C.

This kind of attack, while somewhat unusual in that it explicitly reveals the intruder's identity, may prove harmful for a number of reasons. For example, it deprives C of the possibility to claim later that it communicated with S, since S will not be able to support C's claims (S may not even know about C's existence). If S is a pay server providing some kind of online service in exchange for anonymous "electronic coins" such as eCash [5], I may be able to receive service from S using C's cash. Recall, however, that I can only receive the service if it is not encrypted, which might be the case for large volumes of data.

$C \to S\langle I\rangle$	$C,\ Ver_C,\ Suite_C$
$I \to S$	$\underline{I},\ Ver_C,\ Suite_C$
$S \to I$	$Ver_S,\ Suite_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$I \to C$	$Ver_S,\ Suite_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$C \to S\langle I\rangle$	$\mathrm{sign}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+},$ $\mathrm{sign}_C\{\mathrm{Hash}\,(Secret_C)\}$
$I \to S$	$\underline{\mathrm{sign}_{CA}\{I, V_I\}},\ \{Secret_C\}_{K_S^+},$ $\underline{\mathrm{sign}_I}\{\mathrm{Hash}\,(Secret_C)\}$

⟨Change to negotiated cipher⟩

$S \to I$	$\{\mathrm{Hash}\,(Ver_C, Suite_C, Ver_S,$ $Suite_S)\}_{\mathrm{Master}(Secret_C)}$
$I \to C$	$\{\mathrm{Hash}\,(Ver_C, Suite_C, Ver_S,$ $Suite_S)\}_{\mathrm{Master}(Secret_C)}$
$C \to S\langle I\rangle$	$\{\mathrm{Hash}\,(Ver_C, Suite_C, Ver_S,$ $Suite_S)\}_{\mathrm{Master}(Secret_C)}$
$I \to S$	$\{\mathrm{Hash}\,(Ver_C, Suite_C, Ver_S,$ $Suite_S)\}_{\mathrm{Master}(Secret_C)}$

4.7 Protocol E

D + post-handshake verification of all messages

To fix the bug in Protocol D, the parties verify *all* of their communication after the handshake is complete. Now the intruder may not re-transmit C's messages to S, because C's *Hello* message contained C, while the *Hello* message received by the server contained I. The discrepancy will be detected in post-handshake verification.

$C \to S$	$C,\ Ver_C,\ Suite_C$
$S \to C$	$Ver_S,\ Suite_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$C \to S$	$\mathrm{sign}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+},$ $\mathrm{sign}_C\{\mathrm{Hash}\,(Secret_C)\}$

⟨Change to negotiated cipher⟩

$S \to C$	$\{\mathrm{Hash}\,(Messages)\}_{\mathrm{Master}(Secret_C)}$
$C \to S$	$\{\mathrm{Hash}\,(Messages)\}_{\mathrm{Master}(Secret_C)}$

Attack on E

I observes a run of the protocol and records all of C's messages. Some time later, I initiates a new run of the protocol, ostensibly from C to S, and replays recorded C's messages in response to messages from S. Even though I is unable to read the recorded messages, it manages to convince S that the latter is talking to C, even though C did not initiate the protocol.

$C \to S$	$C,\ Ver_C,\ Suite_C$
$S \to C$	$Ver_S,\ Suite_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$C \to S$	$\mathrm{sign}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+},$ $\mathrm{sign}_C\{\mathrm{Hash}\,(Secret_C)\}$

⟨Change to negotiated cipher⟩

$S \to C$	$\{\mathrm{Hash}\,(Messages)\}_{\mathrm{Master}(Secret_C)}$
$C \to S$	$\{\mathrm{Hash}\,(Messages)\}_{\mathrm{Master}(Secret_C)}$

Next run of the protocol ...

$I \to S$	$C,\ Ver_C,\ Suite_C$
$S \to C\langle I\rangle$	$Ver_S,\ Suite_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$I \to S$	$\mathrm{sign}_{CA}\{C, V_C\},\ \{Secret_C\}_{K_S^+},$ $\mathrm{sign}_C\{\mathrm{Hash}\,(Secret_C)\}$

⟨Change to negotiated cipher⟩

$S \to C\langle I\rangle$	$\{\mathrm{Hash}\,(Messages)\}_{\mathrm{Master}(Secret_C)}$
$I \to S$	$\{\mathrm{Hash}\,(Messages)\}_{\mathrm{Master}(Secret_C)}$

4.8 Protocol F

E + nonces

By adding random nonces to each run of the protocol, SSL 3.0 ensures that there are always some differences between independent runs of the protocol. The intruder is thus unable to replay verification messages from one run in another run.

$C \rightarrow S$	$C,\ \mathit{Ver}_C,\ \mathit{Suite}_C,\ N_C$
$S \rightarrow C$	$\mathit{Ver}_S,\ \mathit{Suite}_S,\ N_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$C \rightarrow S$	$\mathrm{sign}_{CA}\{C, V_C\},\ \{\mathit{Secret}_C\}_{K_S^+},$ $\mathrm{sign}_C\{\mathrm{Hash}\,(\mathit{Secret}_C)\}$

⟨Change to negotiated cipher⟩

$S \rightarrow C$	$\{\mathrm{Hash}\,(\mathit{Messages})\}_{\mathrm{Master}(\mathit{Secret}_C)}$
$C \rightarrow S$	$\{\mathrm{Hash}\,(\mathit{Messages})\}_{\mathrm{Master}(\mathit{Secret}_C)}$

Attack on F

The exact semantics of the verification messages exchanged after switching to the negotiated cipher (i.e., *Finished* messages in the SSL terminology) is somewhat unclear. Section 5.6.9 of [6] states: "No acknowledgment of the finished message is required; parties may begin sending encrypted data immediately after sending the finished message. Recipients of finished messages must verify that the contents are correct." The straightforward implementation of this definition led Murφ to discover the following attack on Protocol F:

1. I modifies the *Hello* messages, changing the legitimate parties' cryptosuites so as to force them into choosing a weak public-key encryption algorithm for the secret exchange.

2. I records the weakly encrypted Secret_C as it is being transmitted from C to S.

3. After C and S switch to the negotiated cipher, I delays their verification messages indefinitely, preventing them from discovering the attack on cryptosuites and gaining extra time to crack the public-key encryption algorithm and learn Secret_C.

4. Once the secret is learned, I is able to compute the keys and forge verification messages.

Since we did not model weak encryption that can be broken by the intruder, we also did not model the last step of the attack explicitly. Instead, if the model reached the state after the third step, the attack was considered successful.

Note that in the actual SSL 3.0 protocol Secret_C is not used directly as the symmetric key between C and S. It serves as one of the inputs to a hash function that computes the actual symmetric key. Therefore, even if the intruder is able to figure out the symmetric key, this will not necessarily compromise the shared secret Secret_C.

To obtain Secret_C, the intruder has to force the parties into choosing weak public-key encryption for the *secret exchange* message, and then break the chosen encryption algorithm in real-time. This attack can only succeed if both parties support cryptosuites with very weak public-key encryption (e.g., with a very short RSA key). We are not aware of any existing SSL implementations for which this is the case.

4.9 Protocol Z (final)

To prevent the attack on Protocol F, it is sufficient to require that the parties consider the protocol incomplete until they each receive the correct verification message. Murφ did not discover any bugs in the model implemented according to this semantics.

Alternatively, yet another piece of SSL can be added to Protocol F. If the client sends the server a hash of all messages *before* switching to the negotiated cipher, the server will be able to detect an attack on its cryptosuite earlier.

$C \rightarrow S$	$C,\ \mathit{Ver}_C,\ \mathit{Suite}_C,\ N_C$
$S \rightarrow C$	$\mathit{Ver}_S,\ \mathit{Suite}_S,\ N_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
$C \rightarrow S$	$\mathrm{sign}_{CA}\{C, V_C\},\ \{\mathit{Secret}_C\}_{K_S^+},$ $\mathrm{sign}_C\{\mathrm{Hash}\,(\mathit{Messages})\}$

⟨Change to negotiated cipher⟩

$S \rightarrow C$	$\{\mathrm{Hash}\,(\mathit{Messages})\}_{\mathrm{Master}(\mathit{Secret}_C)}$
$C \rightarrow S$	$\{\mathrm{Hash}\,(\mathit{Messages})\}_{\mathrm{Master}(\mathit{Secret}_C)}$

Murφ was used to model Protocol Z with 2 clients, 1 intruder, 1 server, no more than 2 simultaneous open sessions per server, and no more than 1 resumption per session. No new bugs were discovered. However, Murφ found two anomalies in the protocol employed to resume a session.

4.9.1 Protocol Z with resumption: cryptosuite attack

Adding the extra verification message suffices for the full handshake protocol but not for the resumption protocol. When a session is resumed, the parties switch to the negotiated cipher immediately after exchanging *Hello* messages. Therefore, the intruder can alter cryptographic preferences in the *Hello* messages and then delay the parties' *Finished* messages indefinitely, preventing them from detecting the attack. It appears that this attack does not jeopardize the security of SSL 3.0 in practice, since no secret is exchanged in the resumption protocol. In fact, it is not clear to us if the cryptosuites in the *Hello* messages are used at all in the resumption protocol.

4.9.2 Protocol Z with resumption: version rollback attack

In our model of Protocol Z, the participants switch to SSL 2.0 if a version number in the *Hello* messages is different from 3.0. (Since the Internet draft for SSL 2.0 has expired and is not publicly available at the moment, we included a specification of SSL 2.0 in Appendix A.)

The *Finished* messages in SSL 2.0 do not include version numbers or cryptosuites, therefore Protocol Z is susceptible to the attack on cryptographic preferences described in Section 4.5. In the following example, it is assumed for simplicity that client authentication is not used. Also, SSL 2.0 *Hello* messages have a slightly different format than SSL 3.0 *Hello* messages and do not contain explicit version information. To simplify presentation, we assume that the intruder converts a 3.0 *Hello* message into a 2.0 *Hello* message simply by changing the version number.

$C \to S\langle I \rangle$	C, 3.0, $Suite_C$, N_C
$I \to S$	C, $\underline{2.0}$, $\underline{Suite_I}$, N_C
$S \to C\langle I \rangle$	2.0, $Suite_S$, N_S, $\text{sign}_{CA}\{S, K_S^+\}$
$I \to C$	2.0, $\underline{Suite_I}$, N_S, $\text{sign}_{CA}\{S, K_S^+\}$
$C \to S$	$\{Secret_C\}_{K_S^+}$

⟨Change to negotiated cipher⟩

$C \to S$	$\{N_S\}_{\text{Master}(Secret_C)}$
$S \to C$	$\{N_C\}_{\text{Master}(Secret_C)}$

To prevent the version rollback attack, SSL 3.0 clients add their version number to the secret they send to the server. When the server receives a secret with 3.0 embedded in it from a 2.0 client, it can determine that there has been an attack on the client's *Hello* message in which the client's true version number (3.0) was rolled back to 2.0.

However, this defense does not work in the case of session *resumption*. Murφ discovered a version rollback attack on the resumption protocol. The attack succeeds since in the resumption protocol, the client does not send a secret to the server, and the intruder's alteration of version numbers in the *Hello* messages goes undetected.

Strictly speaking, this attack is not a violation of the specification [6], since the latter implicitly allows an SSL session that was established using the 3.0 protocol to be resumed using the 2.0 protocol. However, this attack makes implementations of SSL 3.0 potentially vulnerable to SSL 2.0 weaknesses. Wagner and Schneier [14] reach a similar conclusion in their informal analysis of SSL 3.0.

4.10 Protocol Z vs. SSL 3.0

Figure 1 shows the definition of the SSL 3.0 handshake protocol according to [6]. When several messages from the same party follow each other in the original definition, they have been collapsed into a single protocol step (e.g., *Certificate*, *ClientKeyExchange*, and *CertificateVerify* were joined into *ClientVerify*). The underlined pieces of SSL 3.0 are not in Protocol Z.

Assuming that the cryptographic functions are perfect, the underlined pieces can be removed from the SSL 3.0 handshake protocol without jeopardizing its security. However, they do serve a useful purpose by strengthening cryptography and making brute-force attacks on the protocol less feasible.

For example, recall that the shared secret is not used directly as the symmetric key between C and S. Instead, it is used as input to a (pseudorandom) function that computes the actual shared secret. Therefore, breaking the symmetric cipher will not necessarily compromise the shared secret as it would require inverting two hash functions. To obtain the shared secret, the intruder would have to break public-key encryption in the *ClientKeyExchange* message.

The construction of the keyed hash in *ClientVerify*, *ServerFinished*, and *ClientFinished* messages as Hash $(K, Pad_2, \text{Hash}\,(K, Pad_1, text))$ follows the HMAC method proposed by Krawczyk [7], who proved that adding a secret key to the function makes it significantly more secure even if the actual hash function is relatively weak.

In general, we would like to emphasize that SSL 3.0 contains many security measures that are designed to protect the protocol against cryptographic

`ClientHello`	$C \to S$	$C,\ Ver_C,\ Suite_C,\ N_C$
`ServerHello`	$S \to C$	$Ver_S,\ Suite_S,\ N_S,\ \text{sign}_{CA}\{S, K_S^+\}$
`ClientVerify`	$C \to S$	$\text{sign}_{CA}\{C, V_C\},$ $\{Ver_C, Secret_C\}_{K_S^+},$ $\text{sign}_C\{\text{ Hash }(\underline{\text{Master}(N_C,\ N_S,\ Secret_C) + Pad_2 +}$ $\underline{\text{Hash}}\,(Messages\underline{+C\ +}$ $\underline{\text{Master}(N_C,\ N_S,\ Secret_C)\ + Pad_1))}\,\}$
⟨Change to negotiated cipher⟩		
`ServerFinished`	$S \to C$	$\{\text{ Hash }(\underline{\text{Master}(N_C,\ N_S,\ Secret_C) + Pad_2 +}$ $\underline{\text{Hash}}\,(Messages\ \underline{+\ S\ +}$ $\underline{\text{Master}(N_C,\ N_S,\ Secret_C)\ +}$ $\underline{Pad_1))}\,\}_{\text{Master}(\underline{N_C,\ N_S,}\ Secret_C)}$
`ClientFinished`	$C \to S$	$\{\text{ Hash }(\underline{\text{Master}(N_C,\ N_S,\ Secret_C) + Pad_2 +}$ $\underline{\text{Hash}}\,(Messages\ \underline{+\ C\ +}$ $\underline{\text{Master}(N_C,\ N_S,\ Secret_C)\ +}$ $\underline{Pad_1))}\,\}_{\text{Master}(\underline{N_C,\ N_S,}\ Secret_C)}$

Figure 1. The SSL 3.0 handshake protocol

attacks. Since we modeled an idealized protocol in Murφ under the perfect cryptography assumption, we found SSL 3.0 secure even without these features.

5 Optimizing the intruder model

While one goal of our analysis was "rational reconstruction" of the SSL 3.0 handshake protocol, we were also interested in lessons to be learned about using finite-state analysis to verify large protocols. We were particularly concerned about the potentially very large number of reachable states, given that the SSL handshake protocol consists of 7 steps, and each message sent in a particular step includes several components, each of which can be changed by the intruder under certain conditions.

Our model of the intruder is very simple and straightforward. There is no intrinsic knowledge of the protocol embedded in the intruder, nor does the design of the intruder involve any prior knowledge of any form of attack. The intruder may monitor communication between the protocol participants, intercept and generate messages, split intercepted messages into components and recombine them in arbitrary order. No clues are given, however, as to which of these techniques should be used at any given moment. Therefore, the effort involved in implementing the model of the intruder is mostly mechanical.

The following simple techniques proved useful in reducing the number of states to be explored:

Full knowledge. We assume that every message sent on the network is intercepted by the intruder. Clearly, this assumption does not weaken the intruder. Without it, however, most of the states that Murφ explored were identical as far as the state of the legitimate participants was concerned and the only difference was the contents of the intruder's database. By ensuring that the database is always as full as it can possibly be (i.e., it includes all knowledge that can be extracted from the messages transmitted on the network thus far), we achieved an order of magnitude reduction in the number of reachable states (e.g., from approximately 200,000 to 5,000 for a single run of the protocol).

No useless messages. We optimized our intruder model to only generate messages that are expected by the legitimate parties and that can be meaningfully interpreted by them in their current state. Since at any point in the protocol sequence, each party is expecting only one particular *type* of message, the number of message types the intruder generates at each step does not exceed the number of protocol participants. Still, the number of ways in which the intruder could construct individual messages might be large since messages are generated from the message components (keys, nonces, etc.) stored in the intruder's database.

No useless components. If the intercepted message can be completely recreated from the components already in the intruder's database, the message is discarded.

Flags. Every procedure executed by the protocol participants after receiving a message is guarded by a flag. By changing flag values, we can turn on and off pieces of the protocol, enabling incremental and "what-if" modeling (e.g., what happens if the server does not verify the hashed secret it receives from the client).

Running under Linux on a Pentium-120 with 32MB of RAM, the verifier requires approximately 1.5 minutes to check for the case of 1 client, 1 server, 1 open session, and no resumptions. Less than 5000 states are explored.

The largest instance of our model that we verified included 2 clients, 1 server, no more than 2 simultaneous open sessions per server, and no more than 1 resumption per session. Checking took slightly under 8 hours, with 193,000 states explored.

6 Conclusions

Our study shows that the finite-state enumeration tool Murφ can be successfully applied to complex security protocols like SSL. The analysis uncovered some anomalies in SSL 3.0. (None of these anomalies, however, poses a direct threat to the security of SSL 3.0.) Of these anomalies at least one had slipped through expert human analysis, confirming the usefulness of computer assistance in protocol design.

We are seeking to improve on the current limitations of our approach in three ways. First, modeling a complex cryptographic protocol in the Murφ language is a relatively time-consuming (but straightforward) task. Automatic translation from a high-level protocol description language like CAPSL to Murφ could significantly reduce the human effort for the analysis. Second, often the protocols have very large numbers of reachable states. Thus, we plan to develop techniques that reduce the number of states that have to be explored when analyzing cryptographic protocols. Finally, "low level" properties of the cryptographic functions used in a protocol often cause the protocol to fail, even if it appears to be correct at the high level. We plan to extend our modeling approach to allow detection of such protocol failures by developing more detailed protocol models.

Acknowledgments

We are grateful to the members of our research group for discussions of the SSL protocol, which resulted in a better understanding thereof. We would also like to thank the anonymous reviewers for comments that helped to clarify further details of the SSL protocol.

Appendix A: SSL 2.0

This appendix outlines the SSL 2.0 protocol. In the protocol description below, *SessionId* is a number that identifies a particular session. When the server starts a new session with the client, it assigns it a fresh *SessionId*. When the client wants to resume a previous session, it includes its *SessionId* in the *Hello* message, and the server returns *SessionIdHit* which is the same session number with the "session found" bit set.

New session

Figure 2 shows the basic SSL 2.0 protocol. Notice that this protocol does not protect plaintext transmitted in the *Hello* messages, making the protocol vulnerable to version rollback and cryptographic preferences attacks described in Section 4.5 above.

A description of other weaknesses in SSL 2.0 can be found in `SSL-Talk` FAQ [1].

Resumed session

Figure 3 shows the SSL 2.0 resumption protocol.

Resumed session with client authentication

Figure 4 shows the SSL 2.0 resumption protocol with authentication, where *Authentication type* is the means of authentication desired by the server, N'_S is the server's challenge, *Certificate type* is the type of the certificate provided by the client, *Client certificate* is the actual certificate (e.g., a CA-signed certificate $\text{sign}_{CA}\{C, V_C\}$ for the client's

`ClientHello`	$C \to S$	$C,\ \mathit{Suite}_C,\ N_C,$
`ServerHello`	$S \to C$	$\mathit{Suite}_S,\ N_S,\ \mathrm{sign}_{CA}\{S, K_S^+\}$
`ClientMasterKey`	$C \to S$	$\{\mathit{Secret}_C\}_{K_S^+}$
⟨Change to negotiated cipher⟩		
`ClientFinish`	$C \to S$	$\{N_S\}_{\mathrm{Master}(\mathit{Secret}_C)}$
`ServerVerify`	$S \to C$	$\{N_C\}_{\mathrm{Master}(\mathit{Secret}_C)}$
`ServerFinish`	$S \to C$	$\{\mathit{SessionId}\}_{\mathrm{Master}(\mathit{Secret}_C)}$

Figure 2. SSL 2.0 basic protocol

`ClientHello`	$C \to S$	$C,\ \mathit{Suite}_C,\ N_C,\ \mathit{SessionId}$
`ServerHello`	$S \to C$	$N_S,\ \mathit{SessionIdHit}$
⟨Change to negotiated cipher⟩		
`ClientFinish`	$C \to S$	$\{N_S\}_{\mathrm{Master}(\mathit{Secret}_C)}$
`ServerVerify`	$S \to C$	$\{N_C\}_{\mathrm{Master}(\mathit{Secret}_C)}$
`ServerFinish`	$S \to C$	$\{\mathit{SessionId}\}_{\mathrm{Master}(\mathit{Secret}_C)}$

Figure 3. SSL 2.0 resumption protocol

verification key), and *Response data* is the data that authenticates the client (e.g., signed challenge $\mathrm{sign}_C\{N_S'\}$).

Appendix B: master secret computation

The SSL 3.0 master secret is computed using

$$\begin{aligned}\mathrm{Master}(N_C, N_S, \mathit{Secret}_C) = \\ &\mathrm{MD5}\,(\mathit{Secret}_C + \mathrm{SHA}\,(\text{'}A\text{'} + K))\ + \\ &\mathrm{MD5}\,(\mathit{Secret}_C + \mathrm{SHA}\,(\text{'}BB\text{'} + K))\ + \\ &\mathrm{MD5}\,(\mathit{Secret}_C + \mathrm{SHA}\,(\text{'}CCC\text{'} + K))\ ,\end{aligned}$$

where $K = \mathit{Secret}_C + N_C + N_S$. In most of this paper, the master secret is denoted simply as $\mathrm{Master}(\mathit{Secret}_C)$.

References

[1] Consensus Development Corporation. Secure Sockets Layer discussion list FAQ, `http://www.consensus.com/security/ssl-talk-faq.html`, September 3, 1997.

[2] S. Dietrich. *A Formal Analysis of the Secure Sockets Layer Protocol.* PhD thesis, Dept. Mathematics and Computer Science, Adelphi University, April 1997.

[3] D. L. Dill. The Murφ verification system. In *Computer Aided Verification. 8th International Conference*, pages 390–3, 1996.

[4] D. L. Dill, S. Park, and A. G. Nowatzyk. Formal specification of abstract memory models. In *Symposium on Research on Integrated Systems*, pages 38–52, 1993.

[5] `http://www.digicash.com/ecash/ecash-home.html`.

[6] A. O. Freier, P. Karlton, and P. C. Kocher. The SSL protocol version 3.0. `draft-ietf-tls-ssl-version3-00.txt`, November 18, 1996.

[7] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed-hashing for message authentication. Internet Request For Comments RFC-2104, February 1997.

`ClientHello`	$C \rightarrow S$	C, Suite_C, N_C, $\mathit{SessionId}$
`ServerHello`	$S \rightarrow C$	N_S, $\mathit{SessionIdHit}$
(Change to negotiated cipher)		
`ClientFinish`	$C \rightarrow S$	$\{N_S\}_{\text{Master}(\mathit{Secret}_C)}$
`ServerVerify`	$S \rightarrow C$	$\{N_C\}_{\text{Master}(\mathit{Secret}_C)}$
`RequestCertificate`	$S \rightarrow C$	$\{\mathit{Authentication\ type},\ N'_S\}_{\text{Master}(\mathit{Secret}_C)}$
`ClientCertificate`	$C \rightarrow S$	$\{\mathit{Certificate\ type},\ \mathit{Client\ certificate},\ \mathit{Response\ data}\}_{\text{Master}(\mathit{Secret}_C)}$
`ServerFinish`	$S \rightarrow C$	$\{\mathit{SessionId}\}_{\text{Master}(\mathit{Secret}_C)}$

Figure 4. SSL 2.0 resumption protocol with authentication

[8] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In *2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 1996.

[9] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using Murφ. In *IEEE Symposium on Security and Privacy*, pages 141–51, 1997.

[10] `http://verify.stanford.edu/dill/murphi.html`.

[11] S. Schneider. Security properties and CSP. In *IEEE Symp. Security and Privacy*, 1996.

[12] U. Stern and D. L. Dill. Automatic verification of the SCI cache coherence protocol. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 21–34, 1995.

[13] D. Wagner. Email communication, August 23, 1997.

[14] D. Wagner and B. Schneier. Analysis of the SSL 3.0 protocol. In *2nd USENIX Workshop on Electronic Commerce*, 1996. Revised version of November 19, 1996 available from `http://www.cs.berkeley.edu/~daw/ssl3.0.ps`.

[15] L. Yang, D. Gao, J. Mostoufi, R. Joshi, and P. Loewenstein. System design methodology of UltraSPARC™-I. In *32nd Design Automation Conference*, pages 7–12, 1995.

Certificate Revocation and Certificate Update

Moni Naor* Kobbi Nissim
Dept. of Applied Mathematics and Computer Science
Weizmann Institute of Science
Rehovot 76100, Israel
{naor, kobbi}@wisdom.weizmann.ac.il

Abstract

A new solution is suggested for the problem of certificate revocation. This solution represents Certificate Revocation Lists by an authenticated search data structure. *The process of verifying whether a certificate is in the list or not, as well as updating the list, is made very efficient. The suggested solution gains in scalability, communication costs, robustness to parameter changes and update rate. Comparisons to the following solutions are included: 'traditional' CRLs (Certificate Revocation Lists), Micali's Certificate Revocation System (CRS) and Kocher's Certificate Revocation Trees (CRT).*

Finally, a scenario in which certificates are not revoked, but frequently issued for short-term periods is considered. Based on the authenticated search data structure scheme, a certificate update scheme is presented in which all certificates are updated by a common message.

The suggested solutions for certificate revocation and certificate update problems is better than current solutions with respect to communication costs, update rate, and robustness to changes in parameters and is compatible e.g. with X.500 certificates.

1 Introduction

The wide use of public key cryptography requires the ability to verify the authenticity of public keys. This is achieved through the use of certificates (that serve as a mean for transferring trust) in a *Public Key Infrastructure* (PKI). A certificate is a message signed by a publicly trusted authority (the *certification authority,* whose public key authenticity may be provided by other means) which includes a public key and additional data, such as expiration date, serial number and information regarding the key and the subject entity.

When a certificate is issued, its validity is limited by an expiration date. However, there are circumstances (such as when a private key is revealed, or when a key holder changes affiliation or position) where a certificate must be revoked prior to its expiration date. Thus, the existence of a certificate is a necessary but not sufficient evidence for its validity, and a mechanism for determining whether a certificate was revoked is needed.

A typical application is a credit card system where the credit company may revoke a credit card, temporarily or permanently, prior to its expiration, e.g. when a card is reported stolen or according to its user's bank account balance.

This work focuses on the problem of creating and maintaining efficient authenticated data structures holding information about the validity of certificates. I.e. how to store, update and retrieve authenticated information concerning certificates.

There are three main types of parties involved with certificates:

1. **Certification authority (CA)**: A trusted party, already having a certified public key, responsible for establishing and vouching for the authenticity of public keys, including the binding of public keys to users through certificates and certificate revocation.

   A CA does not provide on-line certificate information services to users. Instead, It updates a directory on a periodic basis.

   A CA issues certificates for users by signing a message containing the certificate serial number, relevant data and an expiration date. The certificate is sent to a directory and/or given to the user himself. The CA may revoke a certificate prior to its expiration date.

2. **Directory**: One or more non-trusted parties that get updated certificate revocation information from the CA and serve as a certificate database accessible by the users.

*Research supported by BSF grant no. 94-00032.

3. **User**: A non-trusted party that receives its certificate from the CA, and issues queries for certificate information. A user may either query the validity of other users' certificates (we denote users that query other users' certificates as *merchants*) or, get a proof of the validity of his certificate in order to present it with his certificate (for the latter, the proof must be transferable).

The rest of this paper is organized as follows: In Section 2 we briefly review the solutions we are aware of (CRL, CRS and CRT), memory checkers and incremental cryptographic schemes. In Section 3 we give some basic definitions and the theoretical background and restate the problem in terms of finding efficient *authenticated directories* and, in particular, *authenticated search data structures.* The proposed scheme is described in detail in Section 4 and compared with the other schemes in Section 5. Finally, in Section 6, we consider a model in which a directory is not used for extracting certificates, and certificates are updated periodically. We show how a simple modification of our revocation scheme works in this model.

2 Related work and background

In this section we review the solutions we are aware of, namely *Certificate Revocation List* (CRL [20]), *Certificate Revocation System* (CRS [18]) and *Certificate Revocation Trees* (CRT [16]). We continue by reviewing memory checkers, and incremental cryptographic schemes, relating these problems to certificate revocation, these two sections are included as theoretical background, and are not necessary for understanding the rest of the paper.

2.1 Certificate Revocation List (CRL)

A CRL is a signed list issued by the CA identifying all revoked certificates by their serial numbers. The list is concatenated with a time stamp (as an indication of its freshness) and signed by the CA that originally issued the certificates. The CRLs are sent to the directory on a periodic basis, even if there are no changes, to prevent the malicious replay of old CRLs instead of new CRLs.
As an answer to a query, the directory supplies the most updated CRL (the complete CRL is sent to the merchant).

- The main advantage of the scheme is its simplicity.
- The main disadvantage of the scheme is its high directory-to-user communication costs (since CRLs may get very long). Another disadvantage is that a user may not hold a succinct proof for the validity of his certificate.

A reasonable validity expiration period should be chosen for certificates. If the expiration period is short, resources are wasted reissuing certificates. If the expiration period is long, the CRL may get long, causing high communication costs and difficulties in CRL management. Kaufman *et al.* [15, Section 7.7.3] suggested reissuing all certificates whenever the CRL grows beyond some limit. In their proposal, certificates are marked by a serial number instead of an expiration date. (Serial numbers are incremented for each issued certificate. Serial numbers are not reused even when all certificates are reissued.) The CRL contains a field indicating the *first valid certificate.* When all certificates are reissued, the CRL first valid certificate field is updated to contain the serial number of the first reissued certificate.

2.2 Certificate Revocation System

Micali [18] suggested the Certificate Revocation system (CRS) in order to improve the CRL communication costs. The underlying idea is to sign a message for every certificate stating whether it was revoked or not, and to use an off-line/on-line signature scheme [11] to reduce the cost of periodically updating these signatures.
To create a certificate, the CA associates with each certificate two numbers (Y_{365} and N) that are signed along with the 'traditional' certificate data. For each certificate, the CA chooses (pseudo)randomly two numbers N_0, Y_0 and computes (using a one-way function f) $Y_{365} = f^{365}(Y_0)$ and $N = f(N_0)$. (Actually, a stronger assumption on f is required, e.g. that f is one-way on its iterates, i.e. that given $y = f^i(x)$ it is infeasible to find x' such that $y = f(x')$. This is automatically guaranteed if f is a one-way permutation.)
The directory is updated daily by the CA sending it a number C for each certificate as follows:

1. For a non-revoked certificate the CA reveals one application of f, i.e. $C = Y_{365-i} = f^{365-i}(Y_0)$, where i is a daily incremented counter, $i = 0$ on the date of issue.

2. For a revoked certificate, $C = N_0$.

Thus the most updated value for C serves as a short proof (that certificate x was or was not revoked) that the directory may present in reply to a user query x.

- The advantage of CRS over CRL is in its query communication costs. Based on Federal PKI (Public Key Infrastructure) estimates, Micali [18] showed that although the daily update of the CRS is more expensive than a CRL update, the cost of CRS querying is much lower. He estimated the resulting in 900 fold improvement in total communication costs over CRLs. The exact parameters appear in Section 5.

  Another advantage of CRS is that each user may hold a succinct transferable proof of the validity of his certificate. Directory accesses are saved when users hold such proofs and presents them along with their certificates.

- The main disadvantage of this system is the increase in the CA-to-directory communication (it is of the same magnitude as directory-to-users communication, where the existence of a directory is supposed to decrease the CA's communication). Moreover, since the CA's communication costs are proportional to the directory update rate, CA-to-directory communication costs limit the directory update rate.

  The complexity of verifying that a certificate was not revoked is also proportional to the update rate. For example, for an update once an hour, a user may have to apply the function, f, $365 \times 24 = 8760$ times in order to verify that a certificate was not revoked, making it the dominant factor in verification.

The complexity of the Micali's method of verifying a certificate may be improved as follows. Let h be a collision intractable hash function. To issue a certificate, the CA creates a binary hash tree as follows: The tree leaves are assigned (pseudo)random values. Each internal node v is assigned the value $h(x_1, x_2)$ where x_1, x_2 are the values assigned to v's children. The CA signs the root value and gives it as a part of the certificate, the other tree values (and specifically the (pseudo)random values assigned to its leaves are not revealed). To refresh a certificate the ith time, the CA reveals values of the nodes on the path from the root to leaf $2i$ and their children (Thus, the verifier can check that the nodes are assigned a correct. Note that it is not necessary to explicitly give the values of the nodes on the path since these values may be easily computed given the other values). The path serves as a proof for the certificate validity. Amortizing the number of tree nodes the CA has to send the directory, we get that four nodes are sent to update a user's certificate. We make further use of the hash tree scheme in the following sections.

2.3 Certificate Revocation Trees

Kocher [16] suggested the use of Certificate Revocation Trees (CRT) in order to enable the verifier of a certificate to get a short proof that the certificate was not revoked. A CRT is a hash tree with leaves corresponding to a set of statements about certificate serial number X issued by a CA, CA_x. The set of statements is produced from the set of revoked certificates of every CA. It provides the information whether a certificate X is revoked or not (or whether its status is unknown to the CRT issuer). There are two types of statements: specifying ranges of unknown CAs, and, specifying certificates range of which only the lower certificate is revoked. For instance, if CA_1 revoked two certificates, $X_1 < X_2$, than one of the statements is:

$$\textit{if } CA_x = CA_1 \textit{ and } X_1 \leq X < X_2 \textit{ then } X \textit{ is revoked iff } X = v$$

To produce the CRT, the CRT issuer builds a binary hash tree [17] with leaves corresponding to the above statements.
A proof for a certificate status is a path in the hash tree, from the root to the appropriate leaf (statement) specifying for each node on the path the values of its children.

- The main advantages of CRT over CRL are that the entire CRL is not needed for verifying a specific certificate and that a user may hold a succinct proof of the validity of his certificate.

- The main disadvantage of CRT is in the computational work needed to update the CRT. Any change in the set of revoked certificates may result in re-computation of the *entire* CRT.

2.4 Checking the correctness of memories

Blum *et al.* [3] extended the notion of program checking to programs which store and retrieve data from unreliable memory. In their model, a data structure resides in a large memory controlled by an adversary. A program interacts with the data structure via a *checker*. The checker may use only a small reliable memory and is responsible for detecting errors in the data structure behavior while performing the requested operations. An error occurs when a value returned by the data structure does not agree with the corresponding value entered into the data structure. Blum *et al.* showed how to construct an online checker for RAMs using a variant of Merkle's hash-tree authentication scheme for digital signatures [17]. They used universal one way hash functions [19]. [1]

Certificate revocation may be regarded as a variant of memory checking. As in memory checking, an unreliable memory is used for storing and retrieving data. The difference is that in memory checking the same program writes into and reads from memory via the checker, whereas in certificate revocation there exist two distinct non-communicating programs. One program (the CA) writes into an unreliable memory (the directory), the other (a user) reads from the unreliable memory. The fact that the two programs are disconnected raises the need for a mechanism to prevent an adversary from *replaying* old data.

Returning to memory checking, our solution may be regarded as a checker for *dictionaries.*

2.5 Incremental cryptographic schemes

The high CA-to-directory communication in CRS is due to the fact that the CA has to update values not only for certificates whose status was changed since the last update, but for all certificates. Since the fraction of certificates that change status in every update is expected to be small, it would be preferable to use a scheme with communication (and computational work) depending mostly on the number of certificates that change status. Such issues are addressed by *incremental cryptography.*

Incremental cryptography was introduced by Bellare, Goldreich and Goldwasser [4, 5]. The goal of incremental schemes is to quickly update the value of a cryptographic primitive (e.g. hash function, MAC etc.) when the underlying data is modified. Informally, for a given set of modification operators (e.g. insert, delete, replace), in an *ideal* incremental scheme, the computational work needed for updating a value depends only on the number of data modifications.

An ideal incremental authentication scheme based on a 2-3 search tree was suggested in [5]. The scheme is a modification of a standard tree authentication scheme [17] in order to allow efficient insert/delete block operations along with replace block operations. This scheme cannot be used directly for our problem, we modify it for our purposes in Section 4.

3 Authenticated dictionaries

In this section we consider a more abstract version of the problem and translate it to the problem of finding efficient authenticated data structures. The less theoretically inclined readers may skip directly to Section 4 that presents a self-contained description of such a data structure.

Put in an abstract framework, the problem we deal with is the following: find a protocol between a non-trusted prover, P, and a verifier, V for (non)membership in a set S, where S is some finite set defined by a trusted party (the CA) but not known to V. Given an input x, P should prove either $x \in S$ or $x \notin S$, The trusted party may change S dynamically, but it is assumed to be fixed while P and V interact.

The prover has access to a data structure representing S along with some approved public information about S, created by the trusted party prior to setting x. The non-trusted prover should have an efficient procedure for providing on-line a short proof (e.g. of low order polynomial in $|x|, \log|S|$) for the appropriate claim for x.

3.1 Definitions

Let U be a universe and S be a set such that $S \subseteq U$. Let D_S be a data structure representing S.

- A *membership query* is of the form $\langle e \rangle$. The response to the query is a string $\langle a \rangle$ where $a \in \{\text{YES,NO}\}$, corresponding to $e \in S$, $e \notin S$.
- An *authenticated membership query* is of the form $\langle e \rangle$. The response to the query is a string $\langle a, p \rangle$ where $a \in \{\text{YES,NO}\}$ and p is a proof for a authenticated by a CA.
- *Update* operations are of the form
  1. $\langle \textit{Insert}, e \rangle$ where e is an element in $U \setminus S$. The resulting data structure $D_{S'}$ represents the set $S' = S \cup \{e\}$.
  2. $\langle \textit{Remove}, e \rangle$ where e is an element in S. The resulting data structure $D_{S'}$ represents the set $S' = S \setminus \{e\}$.

Definition 3.1 *A* dictionary *is a data structure D_S representing S supporting membership queries and update operations.*

[1] Informally, U is a family of universal one way hash functions if $\forall x$, for f chosen at random from U, it is infeasible to find y such that $f(x) = f(y)$.

Definition 3.2 *An* authenticated dictionary *is a data structure D_S^{au} representing S supporting authenticated membership queries and update operations.*

In our model, the set S is known both to the CA and the prover, P, but not to the verifier, V. The CA controls S and supplies P with the information needed to create an authenticated dictionary representing S.

Since an authenticated dictionary is dynamic, a mechanism for proving that an authenticated proof is updated is needed. Otherwise, a dishonest directory may replay old proofs. In our model we may assume either that CA updates occur at predetermined times, or, that the user issuing a query knows when the most recent update occurred. In any case, the verifier should be able to check the freshness of a proof p.

The parameters we are interested in regarding authenticated dictionaries are:

- Computational complexity:
    1. The time and space needed to authenticate the dictionary, i.e. creating and updating it.
    2. The time needed to perform an authenticated membership query.
    3. The time needed to verify the answer to an authenticated membership query.
- Communication complexity:
    1. The amount of communication (CA to prover) needed to update the dictionary.
    2. The length of a proof p for an authenticated membership query.

3.2 Implementing authenticated dictionaries

For a small universe U, one can afford computational work proportional to $|U|$. There are two trivial extremes with respect to the number of signed messages, the computation needed in authentication and verification, and the prover to verifier communication complexity:

- For every $e \in U$ the CA signs the appropriate message $e \in U$ or $e \notin U$. To update D_s, $|U|$ signatures are supplied, regardless the number of changed elements in D_S. An example of this solution is the certificate revocation system reviewed in Section 2.2.
- The CA signs a message $M = \sigma_1, \sigma_2, \ldots, \sigma_{|U|}$ where for every $u_i \in U$, σ_i indicates whether $u_i \in S$.

If S is expected to be small, two simple solutions are:

- The CA signs intervals of elements not in S. Such an interval is a pair (s_1, s_2) satisfying $s \notin S$ for all $s_1 \leq s \leq s_2$.
- The CA signs a message M containing a list of every $s \in S$. An example is the certificate revocation lists reviewed in Section 2.1.

In all the solutions above, the messages are signed by the CA and include the time of update.

In the following we describe a generic method for creating an authenticated dictionary D_S^{au} from a dictionary D_S. The overhead in membership queries and update operations is roughly a factor of $\log |D_S|$. In this construction we use a *collision intractable hash function.*

Definition 3.3 *A collision intractable hash function is a function $h()$ such that it is computationally infeasible to find $y \neq x$ satisfying $h(x) = h(y)$.*

Let D_S be a dictionary of size $|D_S|$ representing a set S. Let T_q, T_u be the worst case time needed to a compute membership query or an update operation respectively.

Let h be a collision intractable hash function, and T_h be the time needed to compute h on instances of U. Consider the representation $D_S = (\delta_1, \delta_2, \ldots)$ of S. This may be e.g. a list of all the variables' values composing D_S, or, the way D_S is represented in random access memory. The authenticated dictionary D_S^{au} contains D_S plus a hash tree [17] whose nodes correspond to $\delta_1, \delta_2, \ldots$, and a message signed by the CA containing the tree root value and the time of update.

The hash tree is constructed as follows: A balanced binary tree is created whose leaves are assigned the values $\delta_1, \delta_2, \ldots$. Each internal node v is then assigned a value $h(x_1, x_2)$ where x_1, x_2 are the values assigned to v's children.

- A membership query is translated to an authenticated membership query by supplying a proof for every item δ_i of D_S accessed in the computation. The proof consists of the values of all the nodes on the path from the root to position i and their children. The complexity of an authenticated membership query is thus $O(T_q \cdot T_h \cdot \log |D_S|)$.

- After an update, the portion of the hash tree corresponding to elements δ_i that were changed is re-computed (i.e. all paths from a changed element δ_i to the root). The complexity of an update operation is thus $O(T_u \cdot T_h \cdot \log|D_S|)$.

3.3 Authenticated search data structures

The general method of Section 3.2 for creating dictionaries has a logarithmic (in $|D_s|$) multiplicative factor overhead. The reason is that the internal structure of D_S was not exploited in the authentication/verification processes.

Our goal is to create authenticated dictionaries based on efficient search data structures that save the logarithmic factor overhead. We denote these as *authenticated search data structures.*

CRTs reviewed in Section 2.3 save this logarithmic factor in membership query complexity (but not in update where the amount of computational work is not a function of the number of changes but of the size of the revocation list). In Section 4.1 we show how to create authenticated search data structures based on search trees. An interesting open question is how to construct more efficient authenticated search data structures, e.g. based on hash tables, where membership query is processed in roughly $O(1)$.

4 The proposed scheme

The proposed scheme is closer in spirit to CRL and CRT than to CRS, since it maintains a list of only the revoked certificates. It reduces the CA's communication and actually makes it feasible to update the directory periodically many times a day, achieving a very fine update granularity. The revoked certificates list is maintained in an authenticated search data structure. The benefits of this construction are:

1. It is easy to check and prove whether a certain certificate serial number is in the list or not, without sending the complete list.
2. List update communication costs are low.

The underlying idea is to imitate a search in a data structure constructing a proof for the result during the search. For that, we combine a hash tree scheme (as in [17]) with a sort tree, such that tree leaves correspond to revoked certificates, sorted according to their serial numbers. Both proving that a certificate is revoked and that a certificate is not revoked reduce to proving the existence of certain leaves in the tree:

- Proving that a certificate was revoked is equivalent to proving the existence of a leaf corresponding to it.
- Proving that a certificate was not revoked is equivalent to proving the existence of two certificates corresponding to two neighboring leaves in the tree. One of these certificates has a lower serial number than the queried certificate, and the other has a higher serial number. (We modify this to a proof of the existence of a single leaf at the end of this section.)

4.1 An authenticated search data structure

We maintain a 2-3 tree with leaves corresponding to the revoked certificates' serial numbers in increasing order. (In a 2-3 tree every interior node has two or three children and the paths from root to leaves have the same length. Testing membership, inserting and deleting a single element are done in logarithmic time, where the inserting and deleting of an element affect only the nodes on the insertion/deletion path. For a detailed presentation of 2-3 trees see [1, pp. 169-180].) The property of 2-3 trees that we use is that membership queries, insertions and deletions involve only changes to nodes on a search path. I.e. every change is local and the number of affected paths is small.

The tree may be created either by inserting the serial numbers of the revoked certificates one by one into an initially empty 2-3 tree, or, by sorting the list of serial numbers and building a degree 2 tree with leaves corresponding to the serial numbers in the sorted list (because the communication complexity is minimal when the tree is of degree 2).

Every tree node is assigned a value according to the following procedure:

- Each leaf stores a revoked certificate serial number as its value.
- The value of an internal node is computed by applying a *collision intractable* hash function $h()$ to the values of its children.

The tree update procedure is as follows:

1. Delete each expired certificate serial number from the 2-3 tree, updating the values of the nodes on the deletion path.

2. Insert each newly revoked certificate serial number into the tree, updating the values of the nodes on the insertion path.

During tree update some new nodes may be created or some nodes may be deleted due to the balancing of the 2-3 tree. These nodes occur only on the search path for an inserted/deleted node.
The certification authority vouches for the authenticity of the data structure by signing a message containing the tree root value and the tree height. A proof that there exists a leaf in the tree with a certain value consists of all node values on a path (of length equal to the tree height) from the root to a leaf, plus the values of these nodes children. The proof is verified by checking the values of the tree nodes values on the given path and its length. Finding a fallacious proof for the existence of a leaf in the tree amounts to breaking the collision intractability of h.

Remark 4.1 *Possible choices for h include the more efficient MD4 [22], MD5 [23] or SHA [21] (collisions for MD4 and for the compress function of MD5 were found by Dobbertin [9, 10]) and functions based on a computational hardness assumption such as the hardness of discrete log [8, 7, 4]* [2] *and subset-sum [14, 12] (these are much less efficient).*

Remark 4.2 *Note that an explicit serial number is not needed. Instead, any string that is easily computed from the certificate (e.g. hash of the certificate) may be used.*

Remark 4.3 *It is possible to use a family of universal one-way hash functions, U, instead of collision intractable hash functions by letting every internal node, v, hold also an index of a function $h \in U$. The function h is randomly chosen whenever v lies in a deletion or insertion path. The value of a node is computed by applying h to the values of its children concatenated with their hash function indices. A motivation for using universal one-way hash functions instead of collision intractable hash functions is the successful attacks on MD4 and MD5 [9, 10]. Since universal one-way hash functions are not susceptible to birthday attacks, their application may result in a smaller increase in communication and storage costs with respect to collision intractable functions. Bellare and Rogaway [6] discuss methods for creating practical universal one-way hash functions.*

Remark 4.4 *The scheme may be used also for online revocation checking of certificates (where the latency between certificate revocation and CRL update is reduced). As the result of a query, the on-line service is supposed to return the current certificate status.*
In general, on-line revocation checking requires the certificate validator to be trusted (where in off-line checking, the directory could be a non-trusted party). In practice, it is enough that the certificate validator honestly informs a user about the last time it was updated by the CA (and may be dishonest with respect to other information). then it is not needed for the CA to update it only on predetermined times, and the CA may update the directory whenever the status of a certificate is changed. Even if such an assumption is not plausible, the CA may use the authenticated search data structure to reduce the number of signatures it has to compute, since a signature has to be computed only when a certificate status is changed and not for every query.

4.1.1 Other data structures

For a simpler implementation of the authenticated data structure, random treaps [2] may be used instead of 2-3 trees. *Treaps* are binary trees whose nodes are associated with (key, priority) pairs. The tree is a binary search tree with respect to node keys (i.e. for every node the keys in its left (resp. right) subtrees are small (resp. greater) than its key), and a heap with respect to node priorities (i.e. for every node its priority is higher than its descendents' priorities). Every finite set of (key, priority) pairs has a unique representation as a treap. In *random treaps*, priorities are drawn at random from a large enough ordered set (thus, they are assumed to be distinct).
Seidel and Aragon [2] present simple algorithms for membership queries, insert and delete operations with expected time complexity logarithmic in the size of the set S stored in the treap. Random treaps may be easily converted into authenticated search data structures similarly to 2-3 trees. The communication costs of these schemes is similar since the expected depth of a random treap is similar to its 2-3 tree counterpart.

- The main advantage of random treaps is that their implementation is much more simple than the implementation of 2-3 trees.

- A drawback of using random treaps is that their performance is not guaranteed in worst case.

[2] The function is $h(x_1, x_2, x_3) = g_1^{x_1} g_2^{x_2} g_3^{x_3} (\mathrm{mod}\ p)$. Let g be a generator in $\mathbb{Z}_p$, the CA may generate $g_i = g^{a_i}$ and compute h using a single exponentiation by $h = g^{\sum a_i x_i (\mathrm{mod}\ p-1)} (\mathrm{mod}\ p)$.

E.g. some users may (with low probability) get long authentication paths.

- Another drawback is that a stronger assumption is needed with respect to the directory. The analysis of random treaps is based on the fact that the adversary does not know the exact representation of a treap. A dishonest directory with ability to change the status of certificates may increase the computational work and communication costs of the system.

4.2 The scheme

We now give details for the operations of the three parties in the system.

CA operations:

- **Creating certificates:** The CA produces a certificate by signing a message containing certificate data (e.g. user name and public key), certificate serial number and expiration date.

- **Initialization:** The CA creates the 2-3 tree, as above, for the set of initially revoked certificates. It computes and stores the values of all the tree nodes and sends to the directory the (sorted) list of revoked certificates serial numbers along with a signed message containing the tree root value, the tree height and a time stamp.

- **Updating:** The CA updates the tree by inserting/deleting certificates from it. After each insertion/deletion, all affected tree node values are re-computed. To update the directory, the CA sends a difference list (stating which certificates are to be added/deleted from the previous list) to the directory plus a signature on the new root value, tree height and time stamp.

Directory operations:

- **Initialization:** Upon receiving the initial revoked certificates list, the directory computes by itself the whole 2-3 tree, checks the root value, tree height and time stamp, and verifies the CA's signature on these values.

- **Response to CA's update:** The directory updates the tree according to the difference list received from the CA. It re-computes the values for all the affected nodes and checks the root value, tree height and time stamp.

- **Response to users' queries:** To answer a user query the directory supplies the user with the signed root value, tree height and time stamp.

  1. If the queried certificate is revoked, for each node in the path from the root to the leaf corresponding to the queried certificate, the directory supplies the user its value and its children values.
  2. If the queried certificate is not revoked (not in the list), the directory supplies the user the paths to two neighboring leaves ℓ_1, ℓ_2 such that the value of ℓ_1 (resp. ℓ_2) is smaller (resp. larger) than the queried serial number.

  Note that to reduce the communication costs, the directory need not send the node values on the path from root, but only the values of the siblings of these nodes, since the user may compute them by itself.

User operations:

The user first verifies the CA's signature on the certificate and checks the certificate expiration date. Then, the user issues a query by sending the directory the certificate serial number s. Upon receiving the directory's answer to a query, the user verifies the CA's signature on the root value, tree height and time stamp.

1. If the directory claims the queried certificate is revoked, the user checks the leaf to root path supplied by the directory by applying the hash function h.
2. If the directory claims the queried certificate is not revoked, the user checks the two paths supplied by the directory and checks that they lead to two adjacent leaves in the 2-3 tree, with values ℓ_1, ℓ_2 The user checks that $\ell_1 < s < \ell_2$.

In the above scheme, the communication costs of verifying that a certificate was not revoked may be twice the communication costs of verifying that a certificate is in the list. To overcome this, the tree may be built such that every node corresponds to two consecutive serial numbers – thus having to send only one path in either case. Since the number of bits needed for holding the value of a tree node, i.e. the hash function security parameter (ℓ_{hash} in the notation below) is more than twice the bits needed for holding a certificate serial number, this does not influence the tree size.

5 Evaluation

The CA-to-directory communication costs of our scheme are optimal (proportional to the number of changes in the revocation list), enabling high update rates. The proof supplied by the directory is of length logarithmic in the number of revoked certificates. This allows the user to hold a short transferable proof of the validity of his certificate and present it with his certificate (This proof may be efficiently updated, we will make use of this feature in the certificate update scheme of Section 6).
In the following, we compare the communication costs of CRL, CRS and our system (the communication costs of CRT are similar to ours). Basing on this analysis, we show that the proposed system is more robust to changes in parameters, and allows higher update rates than the other.
Other advantages of the proposed scheme are:

- The CA has to keep a smaller secret than in CRS.
- Since CA-to-directory communication is low, the CA may communicate with the directory using a slow communication line secured against breaking into the CA's computer (the system security is based on the ability to protect the CA's secrets).
- Since we base our scheme on a 2-3 tree, there is never a need to re-compute the entire tree to update it. This allows higher update rates than CRT.
- Another consequence of the low CA-to-directory communication is that a CA may update many directories, avoiding bottlenecks in the communication network.

5.1 Communication costs

The parameters we consider are:

- n - Estimated total number of certificates ($n = 3,000,000$).
- k - Estimated average number of certificates handled by a CA ($k = 30,000$).
- p - Estimated fraction of certificates that will be revoked prior to their expiration ($p = 0.1$). (We assume that certificates are issued for one year, thus, the number of certificates revoked daily is $\frac{n \cdot p}{365}$.)
- q - Estimated number of certificate status queries issued per day ($q = 3,000,000$).
- T - Number of updates per day ($T = 1$).
- ℓ_{sn} - Number of bits needed to hold a certificate serial number ($\ell_{sn} = 20$).
- ℓ_{stat} - Number of bits needed to hold the certificate revocation status numbers Y_{365-i} and N_0 ($\ell_{stat} = 100$).
- ℓ_{sig} - Length of signature ($\ell_{sig} = 1,000$).
- ℓ_{hash} - Security parameter for the hash function ($\ell_{hash} = 128$).

Values for $n, k, p, q, T, \ell_{sn}, \ell_{stat}$ are taken from Micali [18], ℓ_{sig} and ℓ_{hash} are specific to our scheme.

CRL costs

- The CRL daily update cost is $T \cdot n \cdot p \cdot \ell_{sn}$ since each CA sends the whole CRL to the directory in each update. An alternative update procedure where the CA sends to the directory only a difference list (which serial numbers to add/remove from the previous CRL) costs $\frac{n \cdot p \cdot \ell_{sn}}{365}$.
- The CRL daily query cost is $q \cdot p \cdot k \cdot \ell_{sn}$ since for every query the directory sends the whole CRL to the querying user.

CRS costs:

- The CRS daily update cost is $T \cdot n \cdot (\ell_{sn} + \ell_{stat})$ since for every certificate the CA sends ℓ_{stat} bits of certificate revocation status.
- The CRS daily query cost is $\ell_{stat} \cdot q$.

The proposed scheme:

- To update the directory, the CA sends difference lists of total daily length of $\frac{n \cdot p \cdot \ell_{sn}}{365} + T \cdot \ell_{sig}$.
- To answer a user's query, the directory sends up to $2 \cdot \log_2(p \cdot k)$ numbers, each ℓ_{hash} bits long, totaling $2 \cdot q \cdot \ell_{hash} \cdot \log_2(p \cdot k)$ bits.

The following table shows the estimated daily communication costs (in bits) according to the three schemes.

	CRL costs	CRS costs	Proposed scheme
Daily update (CA-directory)	$6 \cdot 10^6$	$3.6 \cdot 10^8$	$1.7 \cdot 10^4$
Daily queries (Directory-users)	$1.8 \cdot 10^{11}$	$3 \cdot 10^8$	$7 \cdot 10^9$

As shown in the table, the proposed scheme costs are lower than CRL costs both in CA-to-directory and in directory-to-users communication. The CA-to-directory costs are much lower than the corresponding CRS costs but, the directory-to-user (and thus the over all) communication costs are increased. Note that in practice, due to communication overheads, the difference between CRS and the proposed method in Directory-to-users communication may be insignificant.

5.2 Robustness to changes

Our scheme is more robust to changes in parameters than CRL and CRS. Since these are bound to change in time or due to the specific needs of different implementations, it is important to have a system that is robust to such changes.

Changes will occur mainly in the total number of certificates (n) and the update rate (T). In the proposed method, changes in n are moderated by a factor of p. Changes in T are moderated by the fact that the update communication costs are not proportional to nT but to T. Figure 1 shows how the CA-to-directory update communication costs of the three methods depend on the update rate (all other parameters are held constant). The update communication costs limit CRS to about one update a day (Another factor that limits the update rate is the amount of computation needed by a user in order to verify that a certificate was not revoked). The proposed method is much more robust, even allowing once per hour updates.

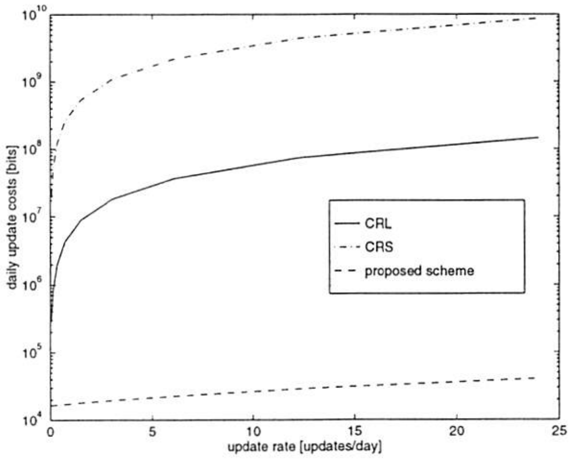


Figure 1: Daily CA-to-directory update costs vs. update rate.

6 A certificate update scheme

Some protocols avoid the need for a revocation system by using short-term certificates. (e.g. micropayments protocols when a certificate owner may cause a limited damage [13]). These certificates are issued daily and expire at the end of the day of issue. Actually, even shorter periods are desired and the main limit is due to the increase in the certification authority computation (certificates for all users have to be computed daily) and communication (certificates should be sent to their owners) short-term certificates cause.

An on-line/off-line digital signature scheme (like CRS) will reduce the computation the CA has to perform, but, it will not reduce significantly the communication costs, since the CA has to send *different* messages to *different* users, making the CA a communication bottleneck. This calls for a solution where the CA performs a simple computation (say, concerning only new users and users whose certificates are not renewed) and sends a *common* update message to *all* users. Using this message, exactly all users with non-revoked certificates should be able to prove the validity of their certificates.

We suggest a simple modification of our certificate revocation scheme that yields an efficient certificate update scheme in which the CA sends the same update message to all users. In this solution we do not assume the existence of a directory with information about all certificates, but of local directories that may hold the latest messages that were sent by the directory.

6.1 The scheme

As before, the scheme is based on a tree of revoked certificates created by the certification authority, presented in Section 4.1. Since there is no way to extract certificates from a directory, every user gets an initial certificate that may be updated using the CA's messages. Specifically, the CA augments every issued certificate with the path proving its validity, this is the only part of the certificate that is updated periodically.

To update all certificates simultaneously, the CA updates its copy of the tree, and publishes the tree paths that where changed since the previous update. Every user holding a non-revoked certificate locates the lowest node, v, on a path that coincides with his path, and updates his path by copying the new node values from v up to the root. All users holding a revoked certificate can not update their path, unless a collision is found for the hash function h.

The information sent by the CA is optimal (up to a factor of ℓ_{hash}). For r insertions/deletions since the previous update, the CA has to publish a message of length $r\ell_{hash} \log n$ bits.
Since the CA communication is reduced, one may use this update scheme for, say, updating certificates once every hour. This may cause some users to lag in updating their certificates, and the local directories should save several latest update messages, and some aggregate updates (combining update messages of a day) enabling uses that lag several days to update their certificates.

Acknowledgments

We thank Omer Reingold for many helpful discussions and for his diligent reading of the paper. We thank the anonymous referees for their helpful comments.

References

[1] A. V. Aho, J. E. Hopcroft, J. D. Ullman. "Data Structures and Algorithms". Addison-Wesley, 1983.

[2] R.G. Seidel., C.R. Aragon "Randomized Search Trees". Proc. 30th Annual IEEE Symp. on Foundations of Computer Science, pp. 540-545, 1989.

[3] M. Blum, W. Evans, P. Gemmell, S. Kannan, M. Naor. "Checking the Correctness of Memories". Algorithmica Vol.12 pp. 225-244, Springer-Verlag, 1994.

[4] M. Bellare, O. Goldreich, S. Goldwasser. "Incremental Cryptography: The Case of Hashing and Signing". Advances in Cryptology - Crypto 94. Ed. Y. Desmedt. Lecture Notes in Computer Science 839, Springer-Verlag, 1994.

[5] M. Bellare, O. Goldreich, S. Goldwasser. "Incremental Cryptography and Application to Virus Protection". Proc. 27th ACS Symp. on Theory of Computing, 1995.

[6] M. Bellare, P. Rogaway. "Collision-Resistant Hashing: Towards Making UOWHFs Practical". Advances in Cryptology - CRYPTO '97, Lecture Notes in Computer Science, Springer-Verlag, 1997.

[7] S. Brands. "An efficient off-line electronic cash system based on the representation problem". CWI Technical Report, CS-R9323, 1993.

[8] D. Chaum, E. van Heijst and B. Pfitzmann. "Cryptographically strong undeniable signatures, unconditionally secure for the signer". Advances in Cryptology - CRYPTO '91, Lecture Notes in Computer Science 576, Springer-Verlag, 1992, pp. 470-484.

[9] H. Dobbertin. "Cryptanalysis of MD4". D. Gollmannn, Ed. Fast Software Encryption, 3rd international workshop. Lecture Notes in Computer Science 1039, Springer-Verlag, pp. 53-69, 1996.

[10] H. Dobbertin. "Cryptanalysis of MD5". Rump session, Eurocrypt 1996.
`http://www.iacr.org/conferences/ec96/rump/index.html`

[11] S. Even, O. Goldreich, S. Micali. "On-Line/Off-Line Digital Signatures". Journal of Cryptology, Springer-Verlag, Vol. 9 pp. 35-67, 1996.

[12] O. Goldreich, S. Goldwasser, and S. Halevi. "Collision-Free Hashing from Lattice Problems". *ECCC*, TR96-042, 1996.
`http://www.eccc.uni-trier.de/eccc/`

[13] A. Herzberg, H. Yochai. "Mini-Pay: Charging per Click on the Web". Proc. 6th International World Wide Web Conference, 1997.
`http://www6.nttlabs.com/`

[14] R. Impagliazzo, M. Naor. "Efficient Cryptographic Schemes Provably as Secure as Subset Sum". Journal of Cryptology, Springer-Verlag, Vol. 9 pp. 199-216, 1996.

[15] C. Kaufman, R. Perlman, M. Speciner. "Network Security. Private Communication in a Public World". Prentice Hall series in networking and distributed systems, 1995.

[16] P. Kocher. "A Quick Introduction to Certificate Revocation Trees (CRTs)".
`http://www.valicert.com/company/crt.html`

[17] R. C. Merkle. "A Certified Digital Signature". Proc. Crypto '89, Lecture Notes in Computer Science 435, pp. 234-246, Springer-Verlag, 1989.

[18] S. Micali. "Efficient Certificate revocation". Technical Memo MIT/LCS/TM-542b, 1996.

[19] M. Naor, M. Yung. "Universal one-way hash functions and their cryptographic applications". Proc. 21st ACM Symp. on Theory of Computing, pp. 33-43, 1989.

[20] U.S. National Institute of Standards and Technology. "A Public Key Infrastructure for U.S. Government unclassified but Sensitive Applications". September 1995.

[21] U.S. National Institute of Standards and Technology. "Secure Hash Standard". Federal Information Processing Standards Publication 180, 1993.

[22] R. Rivest. "The MD4 message-digest algorithm". Internet RFC 1320, 1992.

[23] R. Rivest "The MD5 message-digest algorithm". Internet RFC 1321, 1992.

Attack-resistant trust metrics for public key certification

Raph Levien
Alexander Aiken
University of California, Berkeley
{raph,aiken}@cs.berkeley.edu, http://www.cs.berkeley.edu/~{raph,aiken} [*]

Abstract

This paper investigates the role of *trust metrics* in attack-resistant public key certification. We present an analytical framework for understanding the effectiveness of trust metrics in resisting attacks, including a characterization of the space of possible attacks. Within this framework, we establish the theoretical best case for a trust metric. Finally, we present a practical trust metric based on network flow that meets this theoretical bound.

1 Introduction

Many public key infrastructures have been proposed and some have been deployed. Almost all, however, suffer from a worrisome problem: a compromise of a single key leads to a successful attack on the entire system. For example, an attacker who gains the root key of a certification hierarchy such as PEM [Ken93] can cause anyone in the entire system to accept an arbitrary *forgery* (defined to be an incorrect name/key binding, inserted into the system maliciously).

Recent interest in authentication systems centers on systems requiring a certain number of keys (more than one) to be compromised before a forgery is accepted. This work generally focusses on the concept of a *trust metric*, defined here as a function that computes a *trust value* from a set of digitally signed *certificates*. Informally, a good trust metric ensures that there are really multiple independent sources of certification, and rejects (by assigning low trust values) assertions with insufficient certification.

The previous work raises many questions, including:

- To which kinds of attack is a trust metric resistant?
- Which trust metric is best?
- How well do these trust metrics work?

This paper answers these questions by analysis of the possible attacks against trust metrics. After introducing a certificate system and its mapping to a graph model (Section 2) and more formally defining the notion of a trust metric (Section 3), we present an analytical framework for quantifying the success of various attacks against trust models (Section 4).

To make the analysis tractable, this paper makes two assumptions. First, we assume that most good name/key bindings are accepted. This assumption is not always valid—for high-security applications, it may be desirable to have a very restricted "guest list." However, for applications such as key distribution for IP security [Atk95] and widespread secure e-mail, it is a valid assumption. In such applications, if the trust metric often rejects good name/key bindings, then either connections often fail, or else they must be established insecurely. In either case, a trust metric that accepts most good name/key bindings would seem to be a better alternative.

The second assumption is that the name space is opaque, i.e. no information can be gained from the name itself. Most Internet names have some structure, but relationships between names often

[*] This work was supported in part by the National Science Foundation under grant No. CCR-9457812 and infrastructure grant No. CDA-9401156. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

metric	# nodes needed to mount successful node attack	# nodes needed to mount successful edge attack	# edges needed to mount successful edge attack
shortest path	1	1	1
Maurer	1	1	d
Reiter & Stubblebine	d	d	d
Maxflow	d	d	d
Maxflow-edge	d	αd^2	αd^2
best case	d	αd^2	αd^2

Figure 1: A comparison of the trust metrics.

have little to do with certification relationships, which makes this structure difficult to exploit.

Given this background, the remainder of the paper is devoted to answering the preceding questions, summarized as follows:

- No trust metric can protect against attacks on d keys or more, where d is the minimum number of certifiers on any widely accepted key (Section 5).
- There is an optimal trust metric based on maximum network flow. Such a metric protects against almost any attack on fewer than d keys (Section 6).
- Of previously published trust metrics, the Reiter & Stubblebine [RS97a] trust metric is close to optimal, while the Maurer trust metric [Mau96] is easily attacked (Section 7).

Another contribution of the paper is to distinguish between two different types of attacks on certification systems. The most general form of attack assumes that the attacker is capable of generating arbitrary certificates. This attack corresponds to stealing the secret keys of the victim, and is called a *node attack.*

However, a far more restricted attack is effective against many of the trust metrics and can be easily mounted without stealing secret keys. It suffices to trick owners of the secret keys into certifying that untrustworthy keys are trustworthy; this attack is called an *edge attack.* Fortunately, it is possible to design trust metrics to be more resistant to edge attacks than to node attacks.

Figure 1 is a table that briefly summarizes these results. Here, "shortest path" is the trust metric that simply measures the length of the shortest chain from client to target. "Maurer" is a simplified version of the trust metric proposed by Maurer [Mau96]. "Reiter & Stubblebine" is the bounded vertex disjoint path metric as described in [RS97a]. "Maxflow" is the maximum network flow metric optimized for node attacks (Section 6). "Maxflow-edge" is the maximum network flow metric optimized for edge attacks (Section 6.2). In this table, d is the number of certificates issued for each key in the system, and α is a factor indicating the amount of sharing of certification keys, generally in the range of [0.5..1] (see Section 5.4).

2 Certificates and graphs

The input to the trust model is a set of digitally signed certificates, while the evaluation of the trust model is based on a graph. Depending on the details of the certificate formats themselves, a number of different mappings from certificates to graphs are possible. For the purposes of this paper, we use a simple but realistic certificate format and mapping into a graph.

In this example model, there are *keys*, *names*, and two types of certificates. A *binding certificate* is an assertion of the form "I believe that subject key k is the key belonging to name n", signed by an issuer key. A *delegation certificate* is an assertion of the form "I trust certificates signed by subject key k", again signed by an issuer key. The "I" in these statements refers to the holder of the private key corresponding to the issuer's public

certificate type	issuer	subject
delegation	key_A	key_B
delegation	key_B	key_C
delegation	key_C	key_D
binding	key_D	(key_J, “Jack”)
delegation	key_A	key_E
delegation	key_E	key_F
binding	key_F	(key_J, “Jack”)
delegation	key_E	key_G
delegation	key_G	key_H
delegation	key_H	key_I
binding	key_I	(key_J, “Jack”)

Figure 2: An example set of certificates.

key.

This model corresponds fairly closely to the PGP certificate model [PGP95] extended with “introducer certificates,” not present in any current implementation of PGP but proposed as a future extension. It also resembles an X.509 certification system [X509] with opaque names (as opposed to distiguished names) and cross-certification.

This model is somewhat simplistic compared to real-world certification schemes. For example, it includes no time-dependent behavior such as validity periods or revocation. The model does not distinguish different kinds of certificate issuers, which might be users creating their own certificates or trusted third parties. Some of the assumptions regarding the graph structure are more realistic for user-created certificates.

The mapping is as follows: Each key is a node of the graph. In addition, each (key, name) pair is also a node. Each delegation certificate maps to an edge from the node corresponding to the issuer key to the node corresponding to the subject key. Each binding certificate maps to an edge from the node corresponding to the issuer key to the node corresponding to the (key, name) binding. Figure 2 shows an example set of certificates, and Figure 3 shows the corresponding graph.

Let G be a certificate graph. The nodes of G are either *key nodes* V_k or *target nodes* V_t, thus G can be written as $(V_k \cup V_t, E)$. In the example of Figure 3, $V_t = \{(key\_J, \text{“}Jack\text{”})\}$, and V_k is all the other keys $\{key\_A ... key\_I\}$.

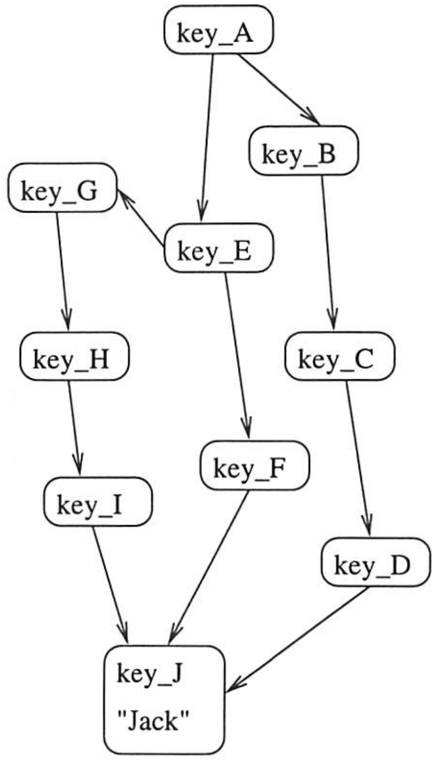


Figure 3: The corresponding graph.

3 Trust models on graphs

This section defines the notion of a trust model as evaluated on graphs. Given a certificate graph G, a *source node* $s \in V_k$, and a *target node* $t \in V_t$, the evaluation of the trust model results in a real number, interpreted as the degree to which the source key should trust the target. Formally, the trust metric is represented as a real-valued function $M(G, s, t)$. The trust model is sensitive only to the structure of the graph, not the names. Thus, the trust metric must give the same value for isomorphic graphs.

In an automated setting, the source s does not use the real number directly, but simply compares it with a threshold $\theta(s)$ to determine whether the target is trustworthy. Formally, a key s *accepts* a target t iff $M(G, s, t) \geq \theta(s)$. Little is lost by expressing the trust metric in terms of acceptance rather than real numbers—by performing multiple experiments with different thresholds, it is possible to reconstruct the numbers with high accuracy. Thus, where it is simpler, we use acceptance to characterize trust metrics. To avoid working directly with real values for the θ function, we often express θ in percentiles, e.g. $\theta(s)$ is set so that s accepts 95% of all targets.

4 Attack models

We consider two different types of attack. In a *node attack,* the attacker is able to generate any certificate from the attacked key. Thus, in a node attack, the attacker may add arbitrary edges (representing delegation or binding certificates) in the graph. This attack is feasible when the attacker obtains the private keying material, for example by stealing a password.

In an *edge attack,* by contrast, the attacker is only able to generate a delegation certificate from the attacked key. This attack is feasible when the attacker is able to convince the owner of the attacked key that an untrustworthy subject key is trustworthy.

We also assume that the attacker is capable of removing arbitrary certificates from any key and generating arbitrary certificates from newly created keys under its control. Removing a certificate is realized by performing a denial-of-service attack on the communication of the certificate to the system requesting it.

For each attack scenario, the number of keys attacked is fixed, counting only keys for which certificates are added.

We now formally present the notion of edge and node attacks on graphs. Given an original certificate graph $G = (V_k \cup V_t, E)$, an attack is represented by a new graph $G' = (V'_k \cup V'_t, E')$. All of the nodes and edges in G are presumed to be "good." The new graph, however, contains at least one new target node $x \in V'_t - V_t$ which represents the forgery.

Given a graph G, the graph G' is a possible node attack on the set of keys T iff:

$$\begin{aligned}&NA((V_k \cup V_t, E), (V'_k \cup V'_t, E'), T) \equiv \\ &\quad \forall s \in V_k - T. \forall t. (s,t) \in E' \Rightarrow (s,t) \in E\end{aligned}$$

This predicate states that G' cannot contain any new edges from nodes in G that are not under attack. Thus, all new edges in G' must come from either a node attack or a new node.

In an edge attack, the edges in G' are further constrained so that no edges from attacked nodes go directly to targets, only to other key nodes.

$$\begin{aligned}&EA'((V_k \cup V_t, E), (V'_k \cup V'_t, E'), T) \equiv \\ &\quad \forall s \in V_k. \forall t \in V'_t. (s,t) \notin E'\end{aligned}$$

$$EA(G, G', T) \equiv NA(G, G', T) \wedge EA'(G, G', T)$$

The last predicate simply states that attacks must satisfy both the NA and EA' predicates.

A consequence of the EA constraint is that none of the edges from attacked nodes point directly to the target x. Thus, in any path from s to x, the attacked node must be at least distance 2 away from x.

4.1 Quantifying the success of an attack

A central contribution of this paper is an analytical framework for quantifying the success of attacks and thus the quality of the trust metric. Trust metrics that make attacks less successful are better.

The success of an attack is most directly measured as the fraction of source keys in V_k accepting the forgery x. Obviously, this fraction depends on the $\theta(s)$ values for the source key. If a source key is tuned to accept very few targets, then it can reject most forgeries as well (in the extreme, accepting no targets is also 100% effective against forgeries). Thus, all measures of success assume fixed target accept rates. For example, it would be reasonable to fix $\theta(s)$ for each source key s so that it accepts 95% of targets. With $\theta(s)$ fixed for each key, it is meaningful to discuss the fraction of source keys that accept a forgery from a specific attack. Formally, the success fraction of an attack G' on a trust metric M is given as:

$$\begin{aligned}&p_{success}(G, G', x) = \\ &\quad |\{s \in V_k | M(G', s, x) \geq \theta(s)\}| / |V_k|, \\ &\quad \text{where } G = (V_k \cup V_t, E)\end{aligned}$$

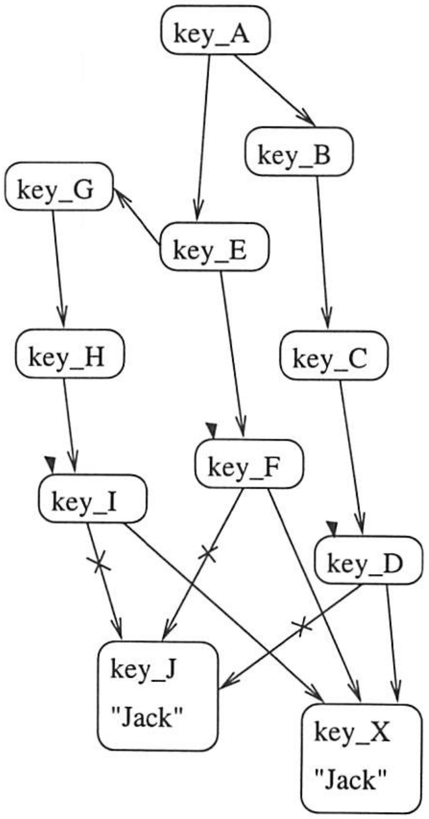


Figure 4: An attack.

For node attacks, it is easier to specify the attack as a set of attacked nodes, rather than an attack graph. The actual graph chosen maximizes the chance of success given the node attack constraint NA.

$$p_{node}(G, x, T) = \max_{G' | NA(G,G',T)} p_{success}(G, G', x)$$

Measuring the success of a particular attack may be useful in some cases, but because the space of individual attacks is so large, it is more useful to characterize trust metrics in terms of their response to classes of attacks. We consider two major classes of attacks: those mounted by randomly choosing nodes to attack, and those mounted by choosing nodes to maximize attack success. In both cases, the attacks are parameterized by the number n of keys attacked.

$$p_{noderand}(G, x, n) = \mathrm{avg}_{T \subseteq V_k \wedge |T|=n}\, p_{node}(G, x, T), \text{ where } G = (V_k \cup V_t, E)$$

$$p_{nodechosen}(G, x, n) = \max_{T \subseteq V_k \wedge |T|=n}\, p_{node}(G, x, T), \text{ where } G = (V_k \cup V_t, E)$$

It is always the case that $p_{noderand}(G, x, n) \leq p_{nodechosen}(G, x, n)$ (i.e. a chosen attack is more effective than a random one). The functions p_{edge}, $p_{edgerand}$, and $p_{edgechosen}$ are all defined analogously, using EA in place of NA.

5 Best case analysis

In this section, we address the question: What is the best possible performance of a trust metric? To make this question tractable, we apply two simplifying assumptions. First, we assume the indegree of every node in the graph is a constant *d*. Second, we assume that most source keys accept most targets in the certificate graph. More formally, $\theta(s)$ of each source s is set so that s accepts the fraction w of all targets. In this analysis, we assume that most clients will want to accept most good targets, so a reasonable value for w would be 99%. Given these assumptions, we prove tight bounds on the best-case performance of any possible graph-based trust metric in resisting attacks.

The assumption of constant indegree *d* instead of a minimum indegree seems to be necessary to avoid overly centralized graphs, i.e. ones in which most nodes have a single edge to a central node. Such a graph meeting a minimum indegree constraint is easily constructed, but has the obvious weakness that an attack on the central node will cause most keys to accept forgeries.

There are several cases, each of which has a slightly different analysis. The general plan is to describe a feasible attack resulting in a graph *G'* isomorphic to the original graph *G* (modulo unreachable nodes), with a forgery x in place of a *victim* v. No trust metric can distinguish the two graphs, thus clients accept the forgery x with the same success fraction as they accepted the victim v.

The idea of the attack is shown graphically in Figure 4, showing how the original graph of Figure 3 is modified—all edges to the victim, in this case (key_J, "Jack"), are removed and replaced with edges to the forgery, here (key_X, "Jack"). The nodes attacked (key_D, key_F, and key_I) are highlighted with wedges.

We consider the following classes of attacks:

- Node attack with a given certificate graph and the attacker chooses the attacked nodes (Section 5.1).
- Node attack with a random certificate graph and the attacked nodes chosen randomly (Section 5.2).
- Node attack with a given certificate graph and the attacker chooses a single attacked node (Section 5.3)
- Edge attack with a given certificate graph and the attacker chooses the attacked nodes (Section 5.4).

In cases where attacks succeed against randomly chosen nodes, the analysis of the chosen node case is not necessary—if the attack works in the former case, it certainly works in the latter. Further, there is a theoretical problem with mounting an attack with randomly chosen attacked nodes against a given certificate graph: the trust metric could just compare the attacked graph against the original graph and reject any certifications if they don't match, an unfair advantage to the trust metric in practice. By showing that the attack works against random graphs, we suggest that it works against most realistic certification graphs.

5.1 Node attack; given certificate graph; attacker chooses nodes

In the first case, we assume a fixed certificate graph with the constraint that the indegree of each node is d. The attack is simple. First, the attacker identifies the node v that is accepted by the largest number of keys. Then, the attacker chooses the d predecessors of that node to attack. Finally, the attacker removes v and its predecessor edges (recall that we assume attackers can remove arbitrary edges, see Section 4), and generates new certificates from each of v's predecessors to the forgery node x.

If all keys accept at least the fraction w of targets, then there must be a target v that is accepted by at least the fraction w of the keys (using the pigeonhole principle). Thus, the success fraction is at least w.

The attack works even if the assumption that sources accept w fraction of all good targets is relaxed. It suffices that there is a single target which is widely accepted by a large fraction of clients, which is very likely in any certification system.

5.2 Node attack; random certificate graph; attacked nodes chosen randomly

Consider the following procedure for generating a random certificate graph. First choose the number of nodes. Second, for each node n, choose d random predecessors (other than n).

The attack is as follows. Choose a node v at random. Remove v and its predecessor edges. Generate d random edges from the remaining nodes to a new node x.

The resulting graph is clearly a member of the set of graphs that may be generated by the random process. Further, it should be clear that the distribution is identical to that of the original random process for constructing graphs. Therefore, no metric can distinguish between the graphs generated randomly and attacked graphs derived from those generated randomly.

5.3 Node attack; given certificate graph; a single node is chosen randomly

For this attack, we assume that the attack is on one node chosen randomly. For certificate graphs with large constant indegree, such an attack cannot be very successful, but for certificate graphs in which a substantial fraction of the nodes have indegree 1 (as is the case with the certificate graph stored on the PGP keyservers [McB96]), the attack can have some success.

The attack only works if the attacked node has a successor v with indegree 1. The attack itself is analogous to those above. Remove v from the graph, generate an edge from the attacked node to x, and generate edges from x to the successors of v. The resulting graph is isomorphic to the original.

Assuming that keys are tuned to accept all good nodes, the success fraction p is equal to the fraction of nodes that have successors of indegree 1 (call this fraction f_1). Assuming that keys accept good targets with probability at least w, then the success fraction is at least $p = (1 - (1 - w)/(1 - f_1))$ (this formula is the lower bound of the probability of the conjunction of two events when the events are not guaranteed to be independent).

5.4 Edge attack; given certificate graph; attacker chooses nodes

Let $pred(v)$ be the predecessors of v, and let $pred^2(v) = \cup_{v' \in pred(v)} pred(v')$. The attack is as follows: find a node v accepted by a sufficiently large number of keys and that has the smallest $pred^2(v)$, which are the nodes attacked. For each node n in $pred(v)$, generate a new node n'. Remove all the edges from $pred^2(v)$ to $pred(v)$, replacing them with edges to the newly created nodes.

The number of nodes that must be attacked is bounded from above by d^2. In practice, there is some α such that there exists a widely accepted node n with $\alpha d^2 = |pred^2(n)|$. In this formulation, α is bounded from above by 1. It is very near one for random graphs, and it is hoped to be fairly high (greater than 0.5, say) for realistic certification graphs.

6 Network flow trust metric

This section presents a trust metric based on maximum network flows over the certificate graph. The analysis of this trust metric shows that its performance almost exactly matches the best case bounds presented above.

The trust metric is defined as follows. As before, let s be the source and t be the target. Each node n in the graph is assigned a capacity $C_{(s,t)}(n) = \max(f_s(\text{dist}(s,n)), g_t(\text{dist}(n,t)))$, where $\text{dist}(s,t)$ is the length of the shortest path through the graph from s to t. The trust metric is parameterized by the exact definitions of f_s and g_t.

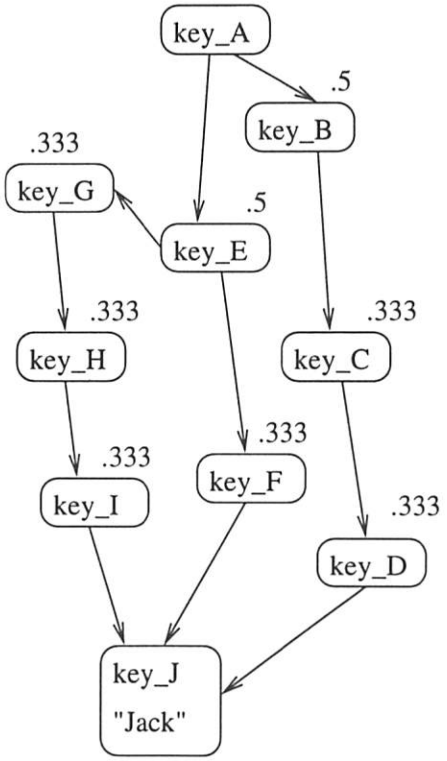


Figure 5: Node capacities for network flow trust metric.

In this section, we present f_s and g_t designed to resist node attacks, but not to do particularly well against edge attacks. Let $succ(s)$ be the set of successors of s. Define

$$f_s(l) = \begin{cases} \max(\frac{1}{d}, \frac{1}{|succ(s)|}) & \text{if } l = 1 \\ \frac{1}{d} & \text{if } l \geq 2 \end{cases}$$

$$g_t(l) = \frac{1}{d}$$

These values are calculated to result in a network maxflow of 1 for most (s, t) pairs in the certificate graph. These values for f_s and g_t guarantee that the number of nodes with $C(n) > 1/d$ is no greater than d.

An example is shown in Figure 5, in which each node (except for the source and target nodes) is annotated with its capacity. In this example, $d = 3$, $f_s(1) = 0.5$, $f_s(2...) = .333$, and $g_t(1...) = 0.333$ (actually, the $d = 3$ constraint is only met for the target node—additional edges needed to satisfy $d = 3$ for all nodes are omitted for brevity).

This example demonstrates why raising node capacities near the source improves security. If the node capacities were constant, they would need to be set at 0.5 to ensure a maxflow of 1. With these settings, most attacks on two nodes succeed. With capacities raised near the source, the only successful two node attack is on $\{key\_B, key\_E\}$. Considering a random attack, as the graph grows the probability of choosing nodes that are all near any given source key becomes much lower. Even considering a chosen node attack, attacking nodes near one source will not in general be effective against other sources.

In another example, assume a random certificate graph as described in Section 5.2 and the unit capacity maxflow metric. Since roughly half of the nodes in such a graph will have outdegree less than d, the threshold must be set at less than d for approximately half of the keys, implying that half of all source keys accept an attack on $d - 1$ keys.

To summarize, in a well-connected graph (i.e. multiple paths between most nodes), with constant capacities, the trust metric is limited by the minimum of the outdegree of the source and the indegree of the target. With capacities increased near the source, the trust metric is limited only by the indegree of the target.

6.1 Analysis: node attacks

The best case analysis shows that a node attack on d nodes is likely to be successful, no matter which trust metric is used. This section shows that, with the network flow trust metric, a node attack on $d - 1$ nodes is unlikely to be successful. Thus, the network flow metric is nearly optimal against node attacks.

We assume an attack where the attacker chooses the nodes to be attacked, as it subsumes the random case (i.e. a trust metric which successfully resists a chosen node attack also resists a randomized attack). We first fix a set S of nodes to attack. Given this set, we analyze the fraction V_k that will accept the forgery. Our goal is to find a tight upper bound on this fraction.

Let us define a node s as *susceptible* to an attack on S iff there exists a node u in S such that $C_{(s,t)}(u) > 1/d$. For any set of nodes S where $|S| = l$ there can be no more than ld susceptible nodes, because the indegree d is fixed, and because the set of susceptible nodes is contained in the predecessors of nodes in S. Thus, at least the fraction $1 - ld/|V_k|$ of all nodes $u \in S$ have $C_{(s,t)}(u) = 1/d$. We know that S is a cut of G' because the NA predicate ensures that all edges from V_k to the new nodes $(V'_k - V_k)$ originate from the attacked keys. Therefore, the total network flow is bounded by $\Sigma_{u \in S}.C_{(s,t)}(u)$, which in this case is no more than l/d. Thus, for at least $1 - ld/|V_k|$ of the source nodes, attacks on less than d nodes fail (i.e. for all $l < d$, $p_{nodechosen}(G, x, l) \leq ld/|V_k|$).

It should be clear that $p_{nodechosen}(G, x, l)$ falls off very quickly as l decreases. In the case where s has at least d successors, s will not accept any attack on less than d nodes.

6.2 Edge attacks

Here are f_s and g_t tuned to resist edge attacks:

$$f_s(l) = \begin{cases} \max(\frac{1}{d}, \frac{1}{|succ(s)|} & \text{if } l = 1 \\ \max(\frac{1}{\alpha d^2}, \frac{1}{|succ^2(s)|}) & \text{if } l = 2 \\ \frac{1}{\alpha d^2} & \text{if } l \geq 3 \end{cases}$$

$$g_t(l) = \begin{cases} \frac{1}{d} & \text{if } l = 1 \\ \frac{1}{\alpha d^2} & \text{if } l \geq 2 \end{cases}$$

Again, we expect there to be a maxmimum flow of 1 for most (s, t) pairs in the graph. For certificate graphs expected to arise in practice, values of α in the range $[0.5..1]$ will lead to a high rate of acceptance. For random graphs, α can be very near unity.

An example is given in Figure 6, in which the immediate successors of s and immediate predecessors of t have capacity 0.333 and all other nodes have capacity .125. The example also shows why a value of 1 for α is not reasonable—in this case, it would cause s to reject t because

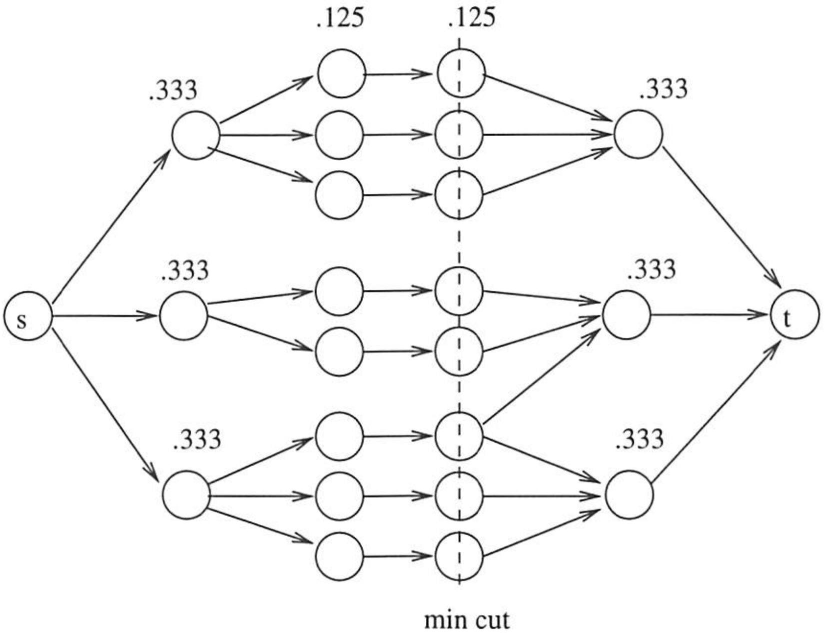


Figure 6: Node capacities for metric tuned for edge attacks.

the predecessors of t share some predecessors. In this example, setting $\alpha = 0.888$ ensures that s accepts t.

6.3 Analysis: edge attacks

The analysis is analogous to that for node attacks, similarly assuming that the attacker chooses the nodes to attack.

Analogous to Section 6.1, let us define a node s as susceptible to an edge attack on S iff there exists a node u for which $C_{(s,t)}(u) > \frac{1}{\alpha d^2}$ and $u \notin \mathit{pred}(t)$. The number of susceptible nodes is no more than $d + d^2$, by of the definition of f_s. Thus, at least the fraction $1 - k(d + d^2)/|V_k|$ of all nodes $u \in S$ have $C_{(s,t)}(u) = \frac{1}{\alpha d^2}$.

The total network flow from s to t is bounded by the minimum cut across the nodes, and hence any cut. Consider the cut across nodes S, which in this case is no more than $k/(\alpha d^2)$. Thus, for at least the fraction $1 - k(d + d^2)/|V_k|$ of all node, edge attacks on less than αd^2 nodes fail.

7 Comparison to related work

Our analytical framework for evaluating trust metrics is new but was inspired by the work of Reiter and Stubblebine [RS97a, RS97b]. Their discussion of the bounded vertex disjoint paths trust model gave one criterion for resistance to attack: with a threshold of k independent paths, attacks on $k - 1$ or fewer nodes will never succeed. In [RS97b], Reiter and Stubblebine show a clear example of a trust metric that is *not* resistant to attack—they demonstrate that in the BBK trust metric [BBK94], an attack on a single node can result in arbitrary manipulation of the trust value. This paper extends their initial work into an analytical framework for comparing trust models in a variety of attack situations.

The general approach of using maximum network flow has been proposed independently by David Johnson, as mentioned in [RS97a]. A different trust metric based on maximum network flow was proposed in [RS97b]. To our knowledge, the idea of increasing the node capacities near the source and target of the query is new. Increasing the capacities near both the source and target has a small effect on node attacks (because the metric is already close to the bound), but does greatly improve resistance to edge attacks—from tacks (in which the attacker can only generate delegation certificates to an untrustworthy party). We have presented a trust metric that is far more resistant to edge attacks than to node attacks. For realistic values, say $d = 10$ (every key is certified by at least 10 others) and $\alpha = 0.5$, an attacker would require 10 keys in a node attack or 50 keys in a edge attack to successfully perpetrate a forgery.

Thus, for an attack-resistant key infrastructure based on trust metrics to be viable, the owner of every key must be willing and able to have a number of people to certify it. Resisting attacks is possible, but increases the cost of certification. Only practical experience with a prototype system can determine whether this tradeoff is worthwhile.

References

[Atk95] R. Atkinson. Security Architecture for the Internet Protocol. RFC 1825, IETF (1995).

[BBK94] T. Beth, M. Borcherding, and B. Klein. Valuation of trust in open netwoks. In D. Gollman, ed., *Computer Security – ESORICS '94* (Lecture Notes in Computer Science 875), pages 3-18, Springer Verlag (1994).

[Ken93] S. Kent, Internet Privacy Enhanced Mail, *Communications of the ACM* 36(8), pp.48-60 (1993).

[Mau96] U. Maurer. Modelling a public-key infrastructure. In E. Bertino, H. Kurth, G. Martella, and E. Montolivo, ed., *Computer Security – ESORICS '96* (Lecture Notes in Computer Science 1146), Springer Verlag (1996).

[McB96] N. McBurnett. PGP web of trust statistics. http://bcn.boulder.co.us/~neal/pgpstat (1996).

[PGP95] P. Zimmermann. The Official PGP User's Guide. MIT Press (1995).

[RS97a] M. Reiter and S. Stubblebine. Path Independence for Authentication in Large-Scale Systems. In *Proceedings of the 4th ACM Conference on Computer and Communications Security* (1997).

[RS97b] M. Reiter and S. Stubblebine. Toward Acceptable Metrics of Authentication. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy* (1997).

[TH92] A. Tarah and C. Huitema. Associating metrics to certification paths. In *Computer Security – ESORICS '92* (Lecture Notes in Computer Science 648), pages 175-189, Springer Verlag (1992).

[X509] International Telegraph and Telephone Consultative Committee (CCITT). *The Directory – Authentication Framework, Recommendatation X.509* (1988).

Appendix A

This appendix presents proofs that the Maurer trust metric becomes consistent with the shortest path and edge-constrained maxflow trust metrics when the edge probability tends to zero or one, respectively.

First, the definition of consistency: Two trust metrics are consistent with each other over a graph G if for all keys, there are no pairs of targets that are ordered inconsistently by the metrics, i.e. M_1 and M_2 are consistent over G iff:

$$\begin{aligned}\forall s, t_1, t_2 . \neg((M_1(G,s,t_1) > M_1(G,s,t_2) \wedge \\ M_2(G,s,t_1) < M_2(G,s,t_2)) \vee \\ (M_1(G,s,t_1) < M_1(G,s,t_2) \wedge \\ M_2(G,s,t_1) > M_2(G,s,t_2)))\end{aligned}$$

Clearly, the existence of a monotonic function f such that $f(M_1(G,s,t)) = M_2(G,s,t)$ is a sufficient condition that M_1 and M_2 are consistent with each other.

Consistency is a weaker relationship than equivalence; it is possible for two trust metrics to be consistent even if they differ in granularity. In the extreme case, the constant trust metric is consistent with all other trust metrics. Thus, the Maurer trust metric may be somewhat "better" than its counterparts because of its finer granularity.

Next, we formally define the trust metric. The trust metric described here is actually a simplified

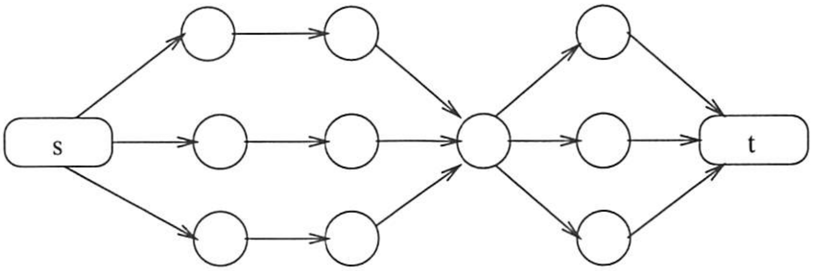


Figure 7: A graph with a bottleneck.

d to αd^2.

Reiter and Stubblebine [RS97a] propose a trust metric based on counting bounded vertex-disjoint paths. In the absence of a path length bound, this trust metric is equivalent to maximum network flow with unit capacities. The question remains: does the imposition of the path length bound improve the metric? Here, we argue that the answer is no.

Assume that trust metric M_1 is maximum flow (without a bound) and a threshold of k_1, and that trust metric M_2 is bounded vertex disjoint paths and a threshold of k_2. For the metrics to be compared directly, source keys must have similar target accept rates. Because the bound causes fewer independent paths to be accepted, $k_1 > k_2$. Thus, there is a class of attacks on k_2 keys accepted by M_2 but not M_1. It is important to stress that this analysis depends on the assumption that the metric accepts most good keys. For metrics that accept only a fraction of the good keys, length bounds in metrics may be useful.

An intriguing metric proposed by [Mau96] assigns probabilities to each edge and performs a randomized experiment on whether the target is reachable from the source. Maurer's paper used a different certificate format and mapping than assumed here. Maurer's graphs have two different kinds of edges, but it is possible to consider a simplified form of his model in which there is a single edge. If we assume that the probabilities on the edges are constant across the entire graph, then as the probability goes to zero, the Maurer trust metric becomes consistent with the shortest path trust metric. As the probability goes to one, the Maurer trust metric becomes consistent with a maximum network flow with unit capacities assigned to edges. Appendix A presents proofs of these statements. Such a trust metric is reasonably effective against edge attacks, but succumbs to an attack on a single node. For example, in the graph shown in Figure 7, there are three edge-independent paths from s to t, but such a graph can be compromised by attacking the single node at the bottleneck.

8 Discussion

We have presented an analytical framework for understanding how a trust metric can be used to resist attacks. Our main assumption is that keys should accept most good targets. We have presented both a best-case theoretical limit on how well trust metrics can perform and a practical trust metric (based on network flows) that meets this limit. Conversely, the fact that the limit is met demonstrates that the attack described in the theoretical analysis (Section 5) is optimal—an attacker should always try to attack the keys nearest the victim node. Under a node attack, the number of nodes that need be attacked is the indegree of the target.

To a first approximation, good trust metrics measure the indegree of the target node. Using a trust metric, then, is only effective if every accepted target has a high indegree. Thus, a trust metric is not particularly helpful on existing certificate graphs such as the PGP key database [McB96], in which about 35% of the keys in the strongly connected subgraph have one predecessor. Each source key has the choice between rejecting a large fraction of the graph or being highly vulnerable to single key forgeries.

We have also distinguished between node attacks (in which the attacker can generate any certificate from a compromised key) and edge at-version of that presented in [Mau96]. One simplification is that the probability associated with each edge is a constant p; in the original metric, the probability can vary depending on user input, or can be encoded in the certificates themselves. The other major simplification is to use a graph with only one type of edge, a consequence of specifying the subject of a delegation certificate by key, rather than name.

We use MM_p to denote the trust metric derived from assigning the probability p to all edges in the graph. We define $MM_p(G, s, t)$ to be the probability that, in randomized experiments in which each edge in G is colored black with probability p, there exists a path from s to t consisting entirely of black edges.

We use *SP* to denote the shortest path metric. $SP(G, s, t)$ is defined as zero minus the shortest path from s to t in graph G, reflecting the fact that shorter paths are to be considered more trustworthy than longer paths.

We use *EDP* to denote the edge disjoint path metric. $EDP(G, s, t)$ is defined as the number of edge-disjoint paths from s to t in graph G.

A.1 Proof for p tending to 0

We prove that MM_p becomes consistent with *SP* as p tends to zero. We show this by presenting a monotonic function f that maps MM_p to *SP*, defined as:

$$f(x) = \max_{x \geq p^{-i}} i, i \text{ integer}$$

We make use of the fact that for monotonic predicates P, $j = \max_{P(i)} i$ iff $P(i)$ and $\neg P(i + 1)$.

Thus, to show that $f(MM_p(G, s, t)) = SP(G, s, t)$, it suffices to show the lower bound $MM_p(G, s, t) \geq p^{SP(G,s,t)}$ and the upper bound $MM_p(G, s, t) < p^{SP(G,s,t)-1}$.

The lower bound is trivial. Let l denote the length of the shortest path from s to t. Consider the subgraph of G containing only a shortest path from s to t. The value of MM_p on this subgraph is computed as p^l, using the series combination rule (Figure 8). Since the Maurer metric is monotonic (i.e. $G' \supseteq G \Rightarrow MM_p(G', s, t) \geq MM_p(G, s, t)$), and $SP(G, s, t) = -l$, the lower bound follows.

To demonstrate the upper bound, we first compute a breadth-first search of G, for each node n assigning $d(n) =$ the length of the shortest path from s to n. We then calculate, for each path length l, an upper bound $u(l)$ on the probability that at least one node n such that $d(n) = l$ is reachable on an all-black path from s, assuming each edge is colored black with independent probability p.

The calculation is by induction. In the base case, $u(0) = 1$ trivially.

In the induction step, if no nodes at distance $l - 1$ are black-reachable, then no nodes at distance l could be black-reachable. Thus, it suffices to compute an upper bound on the conditional probability that at least one node at distance l is black-reachable given that at least one node at distance $l - 1$ is black-reachable. This conditional probability is no greater than $p|E|$. Thus, $u(l) = p|E|u(l - 1) = (p|E|)^l$.

The proof is completed by finding a value of p_0 such that for all $p < p_0$, $u(l) < p^{l-1}$. Solving for equality, we have:

$$(p|E|)^l = p^{l-1}$$

$$p|E|^l = 1$$

$$p = |E|^{-l}$$

By choosing p_0 to solve this equation for the maximum possible value of l ($diam$, the diameter of the graph), we satisfy the inequality for all $p < p_0$ and all $l \leq diam$. Thus, with $p_0 = |E|^{-diam}$ the result is proved.

A.2 Proof for p tending to 1

We prove that MM_p becomes consistent with *EDP* as p tends to one. We show this by presenting a monotonic function f that maps MM_p to *EDP*, defined as:

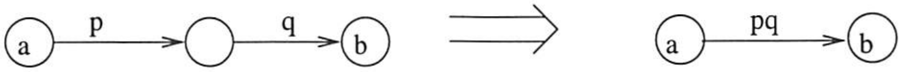


Figure 8: Series combination rule for Maurer metric.

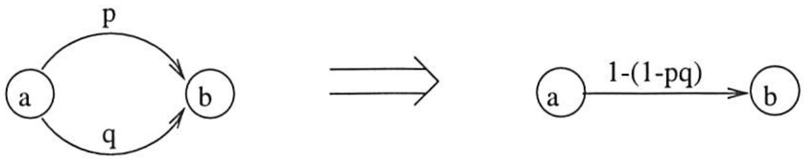


Figure 9: Parallel combination rule for Maurer metric.

$f(x) = \max_{x \geq 1-(1-p^{diam})^i} i$, i integer

As in the shortest path case, we show that $f(MM_p(G,s,t)) = EDP(G,s,t)$ by showing the lower bound $MM_p(G,s,t) \geq 1 - (1 - p^{diam})^{EDP(G,s,t)}$ and the upper bound $MM_p(G,s,t) < 1 - (1 - p^{diam})^{EDP(G,s,t)+1}$.

To show the lower bound, choose the subgraph which contains only k vertex-disjoint paths from s to t. By using the series rule (Figure 8), each path reduces to a single edge of probability p^l, where l is the path length. Because no path can be longer than $diam$ edges, the probability on the edge is bounded from below by p^{diam}. Using the parallel combination rule (Figure 9), these edges reduce to a single edge with probability $1-(1-p^{diam})^k$.

To show the upper bound, consider that because there are only k edge-disjoint paths, there is a set of edges $F, |F| = k$ such that removing those edges blocks all black paths. By computation, the probability of all these edges being removed is $(1-p)^k$. Thus, the probability of an all-black path is bounded from above by $1-(1-p)^k$. To establish the upper bound, we must show that there exists some p_1 such that for all $p > p_1$, $1-(1-p_1)^k < 1-(1-p_1^{diam})^{k+1}$, or equivalently $(1-p_1)^k > (1-p_1^{diam})^{k+1}$.

Because $p^n \geq 1-(1-p)n$ for all $0 \leq p \leq 1$ and all $n > 0$, this inequality is implied by:

$$(1-p_1)^k > ((1-p_1)^{diam})^{k+1}$$

By simple algebra, this is equvalent to:

$$p > 1 - 1/(diam^{k+1})$$

Thus, choosing $p_1 = 1 - 1/(diam^{k+1})$ satisfies the inequality. Because p_1 monotonically increases with k, setting it for the highest possible value of k will satisfy the inequality for all s and t.

These two proofs characterize the behavior of the Maurer metric for values of p near 0 and 1, but not for values in between. As might be expected, simulations with the metric over various certificate graphs indicate that its performance is intermediate between these two cases for intermediate values of p.

Software Generation of Practically Strong Random Numbers

Peter Gutmann

Department of Computer Science
University of Auckland
pgut001@cs.auckland.ac.nz

Abstract

Although much thought usually goes into the design of encryption algorithms and protocols, less consideration is often given to equally important issues such as the selection of cryptographically strong random numbers, so that an attacker may find it easier to break the random number generator than the security system it is used with. This paper provides a comprehensive guide to designing and implementing a practically strong random data accumulator and generator which requires no specialised hardware or access to privileged system services. The performance of the generator on a variety of systems is analysed, and measures which can make recovery of the accumulator/generator state information more difficult for an attacker are presented. The result is an easy-to-use random number generator which should be suitable even for demanding cryptographic applications.

1. Introduction

The best means of obtaining unpredictable random numbers is by measuring physical phenomena such as radioactive decay, thermal noise in semiconductors, sound samples taken in a noisy environment, and even digitised images of a lava lamp. However few computers (or users) have access to the kind of specialised hardware required for these sources, and must rely on other means of obtaining random data.

Existing approaches which don't rely on special hardware have ranged from precise timing measurements of the effects of air turbulence on the movement of hard drive heads [1], timing of keystrokes as the user enters a password [2][3], timing of memory accesses under artificially-induced thrashing conditions [4], and measurement of timing skew between two system timers (generally a hardware and a software timer, with the skew being affected by the 3-degree background radiation of interrupts and other system activity)[5]. In addition a number of documents exist which provide general advice on using and choosing random number sources [6][7][8][9].

Due to size constraints, a discussion of the nature of randomness, especially cryptographically strong randomness, is beyond the scope of this paper. A good general overview of what constitutes randomness, what sort of sources are useful (and not useful), and how to process the data from them, is given in RFC 1750 [10]. Further discussion on the nature of randomness, pseudorandom number generators (PRNG's), and cryptographic randomness is available from a number of sources [11][12][13]. For the purposes of this paper the term "practically strong randomness" has been chosen to represent randomness which isn't cryptographically strong by the usual definitions but which is as close to it as is practically possible.

Unfortunately the advice presented by various authors is all too often ignored, resulting in insecure random number generators which produce encryption keys which are much, much easier to attack than the underlying cryptosystems they are used with. A particularly popular source of bad random numbers is the current time and process ID. This type of flawed generator first gained widespread publicity in late 1995, when it was found that the encryption in Netscape browsers could be broken in around a minute due to the limited range of values provided by this source, leading to some spectacular headlines in the popular press [14]. Because the values used to generate session keys could be established without too much difficulty, even non-crippled browsers with 128-bit session keys carried (at best) only 47 bits of entropy in their session keys [15]. Shortly afterwards it was found that Kerberos V4 suffered from a similar weakness (in fact it was even worse than Netscape since it used `random()` instead of MD5 as its mixing function) [16]. At about the same time, it was announced that the MIT-MAGIC-COOKIE-1 key generation, which created a 56-bit value, effectively only had 256 seed values due to its use of `rand()` (this had been discovered in January of that year but the announcement was delayed to allow vendors to fix the problem) [17].

In a attempt to remedy this situation, this paper provides a comprehensive guide to designing and implementing a practically strong random data accumulator and generator which requires no specialised hardware or access to privileged system services. The result is an easy-to-use random number generator which (currently) runs under BeOS, DOS, the Macintosh, OS/2, Windows 3.x, Windows'95, Windows NT, and Unix, and which should be suitable even for demanding applications.

2. Requirements and Limitations of the Generator

There are several special requirements and limitations which affect the design of a practically strong random number generator. The main requirement (and also limitation) imposed upon the generator is that it can't rely on only one source, or on a small number of sources, for its random data. For example even if it were possible to assume that a system has some sort of sound input device, the signal obtained from it is often not random at all, but heavily influenced by crosstalk with other system components or predictable in nature (one test with a cheap 8-bit sound card in a PC produced only a single changing bit which toggled in a fairly predictable manner).

In addition several of the sources mentioned so far are very hardware-specific or operating-system specific. The keystroke-timing code used in PGP relies on direct access to hardware timers (under DOS) or the use of obscure ioctl's to allow uncooked access to Unix keyboard input, which may be unavailable in some environments, or function in unexpected ways (for example under Windows many features of the PC hardware are virtualised, and therefore provide much less entropy than they appear to; under Unix the user is often not located at the system console, making keystrokes subject to the timing constraints of the `telnet` or `rlogin` session, as well as being susceptible to network packet sniffing). Network sniffing can also reveal other details of random seed data, for example an opponent could observe the DNS queries used to resolve names when `netstat` is run without the `-n` flag, lowering its utility as a potential source of randomness.

Other traps abound. In the absence of a facility for timing keystrokes, mouse activity is often used as a source of randomness. However some Windows mouse drivers have a "snap to" capability which positions the mouse pointer over the default button in a dialog box or window. Networked applications may transmit the client's mouse events to a server, revealing information about mouse movements and clicks. Some operating systems will collapse multiple mouse events into a single meta-event to cut down on network traffic or handling overhead, reducing the input from wiggle-the-mouse randomness gathering to a single mouse move event. In addition if the process is running on an unattended server, there may be no keyboard or mouse activity at all.

In order to avoid this dependency on a particular piece of hardware or operating system, the generator should rely on as many inputs as possible. This is expanded on in "Polling for randomness" below.

The generator should also have several other properties:

- It should be resistant to analysis of its input data. An attacker who recovers or is aware of a portion of the input to the generator should be unable to use this information to recover the generator's state.
- As an extension of the above, it should also be resistant to manipulation of the input data, so that an attacker able to feed chosen input to the generator should be unable to influence its state in any predictable manner. An example of a generator which lacked this property was the one used in early versions of the BSAFE library, which could end up containing a very low amount of entropy if fed many small data blocks such as user keystroke information [18].
- It should be resistant to analysis of its output data. If an attacker recovers a portion of the generator's state, they should be unable to recover any other state information from this (ideally, the generator should never leak any of its state to the outside world). For example recovering generator output such as a session key or PKCS #1 padding for RSA keys should not allow any more of the generator state to be recovered.
- The generator should also take steps to protect its internal state to ensure that it can't be recovered through techniques such as scanning the system swap file for a large block of random data. This is discussed in more detail in "Protecting the randomness pool" below.
- The implementation of the generator should make explicit any actions such as mixing the pool or extracting data in order to allow the conformance of the code to the generator design to be easily checked. This is particularly problematic in the

code used to implement the PGP 2.x random number pool, which (for example) relies on the fact that a pool index value is initially set to point past the end of the pool so that on the first attempt to read data from it the available byte count will evaluate to zero bytes, resulting in no data being copied out and the code dropping through to the pool mixing function. This type of coding makes the correct functioning of the random pool management code difficult to ascertain.

In general, all possible steps should be taken to ensure that the generator state information never leaks to the outside world. Any leakage of internal state which would allow an attacker to predict further generator output should be regarded as a catastrophic failure of the generator. A paper which complements this one and analyses potential generator weaknesses and methods of attack is due to appear in the near future [19].

Given the wide range of environments in which the generator would typically be employed, it is not possible within the confines of this paper to present a detailed breakdown of the nature of, and capabilities of, an attacker. Because of this limitation we take all possible prudent precautions which might foil an attacker, but leave it to end users to decide whether this provides sufficient security for their particular application.

3. The Randomness Pool and Mixing Function

The generator described here consists of two parts, a randomness pool and associated mixing function (the generator itself), and a polling mechanism to gather randomness from the system and add it to the pool (the randomness accumulator). These two parts represent two very distinct components of the overall generator, with the accumulator being used to continually inject random data into the generator, and the generator being used to "stretch" this random data via some form of PRNG. However the PRNG functionality is only needed in some cases. Consider a typical case in which the generator is required to produce a single quantum of random data, for example to encrypt a piece of outgoing email or to establish an SSL shared secret. Even if the transformation function being used in the generator is a completely reversible one such as a (hypothetical) perfect compressor, there is no loss of security because everything nonrandom and predictable is discarded and only the unpredictable material remains as the generator output. Only when large amounts of data are drawn from the system does the "accumulator" functionality give way to the "generator" functionality, at which point a transformation with certain special cryptographic qualities is required (although, in the absence of a perfect compressor, it doesn't hurt to have these present anyway).

Because of the special properties required when the generator functionality is dominant, the pool and mixing function have to be carefully designed to meet the requirements given in the previous section. Before discussing the mixing function used by the generator, it might be useful to examine the types of functions which are used by other generators.

One of the simplest comes from Schneier [8], and consists of a hash function such as MD5 combined with a counter value to create a pseudorandom byte stream generator running in counter mode with a 16-byte output:

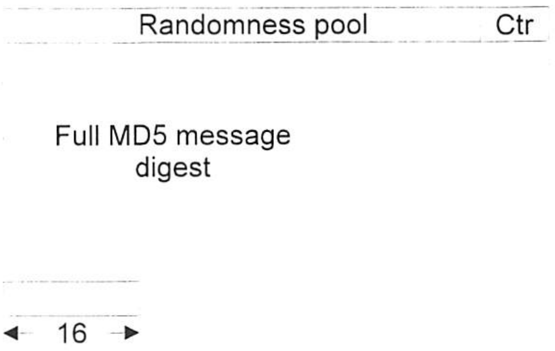


Figure 1: Schneier's Generator

This generator uses the full message digest function rather than just the compression function as most other generators do. It therefore relies on the strength of the underlying hash function for security, and may be susceptible to some form of related-key attack since only one or two bits of input are changed for every block of output produced.

PGP 2.x uses a slightly different method which involves "encrypting" the contents of the pool with the MD5 compression function used as a CFB-mode stream cipher in a so-called "message digest cipher" configuration [20]. The key consists of the previous state of the pool, with the data from the start of the pool being used as the 64-byte input to the compression function. The pool itself is 384 bytes long, although other programs such as CryptDisk and Curve Encrypt for the Macintosh, which also use the PGP random pool management code, extend this to 512 bytes.

The data being encrypted is the 16-byte initialisation vector (IV) which is XOR'd with the data at the current pool position (in fact there is no need to use CFB mode, the generator could just as easily use CBC as there is no need for the "encryption" to be reversible). This process carries 128 bits of state (the IV) from one block to another:

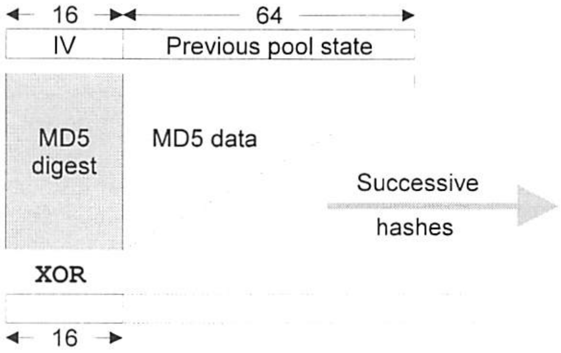


Figure 2: The PGP Mixing Function

The initial IV is taken from the end of the pool, and mixing proceeds until the entire pool has been processed. Newer versions of PGP perform a second pass over the pool for extra security. Once the pool contents have been mixed, the first 64 bytes are extracted to form the key for the next round of mixing, and the remainder of the pool is available for use by PGP. The pool management code allows random data to be read directly out of the pool with no post-processing, and relies for its security on the fact that the previous pool contents, which are being used as the "key" for the MD5 cipher, cannot be recovered. This direct access to the pool is rather dangerous since the slightest coding error could lead to a catastrophic failure in which the pool data is leaked to outsiders. As has been mentioned previously, the correct functioning of the PGP 2.x random number management code is not immediately obvious, making it difficult to spot problems of this nature.

PGP also preserves some randomness state between invocations of the program by storing a nonce on disk which is en/decrypted with a user-supplied key and injected into the randomness pool. This is a variation of method used by the ANSI X9.17 generator which utilises a user-supplied key and a timestamp (as opposed to PGP's preserved state).

PGP 5.x uses a slightly different update/mixing function which adds an extra layer of complexity to the basic PGP 2.x system. In the following pseudocode the arrays are assumed to be arrays of bytes. Where a '32' suffix is added to the name, it indicates that the array is treated as an array of 32-bit words with index values appropriately scaled. In addition the index values wrap back to the start of the arrays when they reach the end:

```
pool[ 640 ], poolPos = 0;
key[ 64 ], keyPos = 0;

addByte( byte )
  {
  /* Update the key */
  key[ keyPos++ ] ^= byte;
  if( another 32-bit word accumulated )
    key32[ keyPos ] ^= pool32[ poolPos ];

  /* Update the pool */
  if( about 16 bits added to key )
    {
    /* Encrypt and perform IV-style block
       chaining */
    hash( pool[ poolPos ], key );
    pool[ next 16 bytes ] ^=
       pool[ current 16 bytes ];
    }
  }
```

This retains the basic model used in PGP 2.x (with a key external to the pool being used to mix the pool itself), but changes the encryption mode from CFB to CBC, and adds feedback between the pool and the MD5 key data. The major innovation in this generator is that the added data is mixed in at a much earlier stage than in the PGP 2.x generator, being added directly to the key (where it immediately affects any further MD5-based mixing) rather than to the pool. The feedback of data from the pool to the key ensures that any sensitive material (such as a user passphrase) which is added isn't left lying in the key buffer in plaintext form.

Once enough new data has been added to the key, the resulting key is used to "encrypt" the pool using MD5, ensuring that the pool data which was fed back to mask the newly-added keying material is destroyed. In this way the entire pool is encrypted with a key which changes slightly for each block rather than a constant key, and the encryption takes place incrementally instead of the using monolithic update technique preferred by other generators.

Another generator inspired by the PGP 2.x one is the Unix /dev/random driver [21], of which a variant also exists for DOS. The driver works by accumulating information such as keyboard and mouse timings and data, and hardware interrupt and block device timing information, which is supplied to it by the kernel. Since the sampling occurs during interrupt processing, it is essential that the mixing of the sample data into the pool be as efficient as possible. For this reason the driver uses a CRC-like mixing function in place of the traditional hash function to mix the data into the pool, with hashing only being done when data is extracted from the pool.

On extracting data the driver hashes successive 64-byte blocks of the pool using the compression function of MD5 or SHA-1, mixes the resulting 16 or 20-byte hash back into the pool, hashes the first 64 bytes of pool one more time to obscure the data which was fed back to the pool, and returns the final 16 or 20-byte hash to the caller. If more data is required, this process is iterated until the pool read request is satisfied. The driver makes two devices available, `/dev/random` which estimates the amount of entropy in the pool and only returns that many bits, and `/dev/urandom` which uses the PRNG described above to return as many bytes as the caller requests.

The function we use improves on the basic mixing function by incorporating far more state than the 128 bits used by the PGP code. The mixing function is again based on a one-way hash function (in which role MD5 or SHA-1 are normally employed) and works by treating a block of memory (typically a few hundred bytes) as a circular buffer and using the hash function to process the data in the buffer. Instead of using the full hash function to perform the mixing, we only utilise the central 16+64→16 byte or 20+64→20 byte transformation which constitutes the hash function's compression function and which is somewhat faster than using the full hash.

Assuming the use of MD5, which has a 64-byte input and 16-byte output, we would hash the 16+64 bytes at locations *n*-16...*n*+63 and then write the resulting 16-byte hash to locations *n*...*n*+15 (the chaining which is performed explicitly by PGP is performed implicitly here by including the previously processed 16 bytes in the input to the hash function). We then move forward 16 bytes and repeat the process, wrapping the input around to the start of the buffer when the end of the buffer is reached. The overlapping of the data input to each hash means that each 16-byte block which is processed is influenced by all the surrounding bytes:

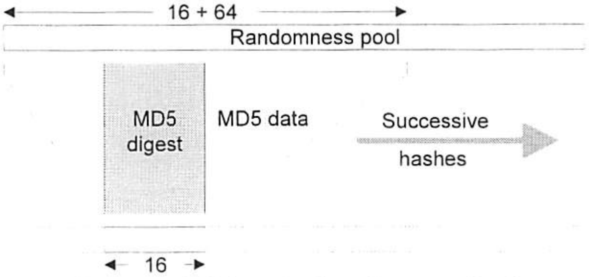


Figure 3: Mixing the Randomness Pool

This process carries 640 bits of state information with it, and means that every byte in the buffer is directly influenced by the 64 bytes surrounding it and indirectly influenced by every other byte in the buffer (although it can take several iterations of mixing before this indirect influence is felt, depending on the size of the buffer). This is preferable to alternative schemes which involve encrypting the data with a block cipher using block chaining, since most block ciphers carry only 64 bits of state along with them.

The pool management code keeps track of the current write position in the pool. When a new data byte arrives, it is added to the byte at the current write position in the pool, the write position is advanced by one, and, when the end of the pool is reached, the entire pool is remixed using the mixing function described above. Since the amount of data which is gathered by the randomness-polling described in the next section is quite considerable, we don't need to perform the input masking which is used in the PGP 5.x generator because a single randomness poll will result in many iterations of pool mixing as all the polled data is added. The pool mixing code does however provide a mechanism to manually force a pool remix in case this is required.

Data removed from the pool is not read out in the byte-by-byte manner in which it is added. Instead, an entire key is extracted in a single block, which leads to a security problem: If an attacker can recover one of the keys, comprising *m* bytes of an *n*-byte pool, the amount of entropy left in the pool is only *n*-*m* bytes, which violates the design requirement that an attacker be unable to recover any of the generator's state by observing its output. This is particularly problematic in cases such as some discrete-log based PKC's in which the pool provides data for first public and then private key values, because an attacker will have access to the output used to generate the public parameters and can then use this output to try to derive the private value(s).

One solution to this problem is to use a generator such as the ANSI X9.17 generator [22] to protect the contents of the pool (in fact some newer versions of PGP do just that, and attach an X9.17-like generator which uses IDEA instead of triple DES to the internal random number pool for use in generating random data for certain applications). In this way the key is derived from the pool contents via a one-way function.

The solution we use is slightly different. What we do is first mix the pool to create the key, then invert every bit in the pool and mix it again to create the new pool, although it may be desirable to tune the operation used to transform the input to the hash function depending on the hash function being used. For example SHA performs a complex XOR-based "key schedule" on the input data, which could potentially lead to problems if the transformation consists of XOR-ing each input word with 0xFFFFFFFF. In this case it might be preferable to use some other form of operation such as a rotate and XOR, or the CRC-type function used by the `/dev/random` driver. If the pool were being used as the key for a DES-based mixing function, it would be necessary to adjust for weak keys; other mixing methods might require the use of similar precautions.

This method should be secure provided that the hash function we use meets its design goal of preimage resistance and is a random function (that is, no polynomial-time algorithm exists to distinguish the output of the function from random strings). The resulting generator is remarkably similar to the triple-DES based ANSI X9.17 generator, which functions as follows:

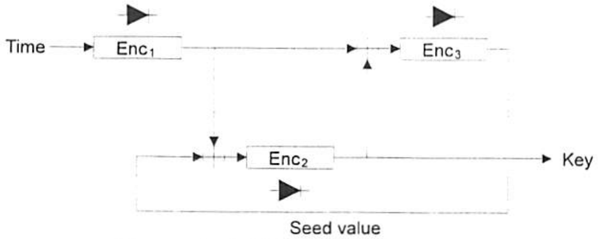


Figure 4: ANSI X9.17 Generator

In the X9.17 generator the encryption step labelled Enc_1 ensures that the timestamp is spread over 64 bits and avoids the threat of a chosen-timestamp attack (for example setting it to all-zero or all-one bits), the Enc_2 step acts as a one-way function for the generated encryption key, and the Enc_3 step acts as a one-way function for the seed value/internal state. The generator presented here replaces the keyed triple-DES operations with an unkeyed one-way hash function which has the same effect:

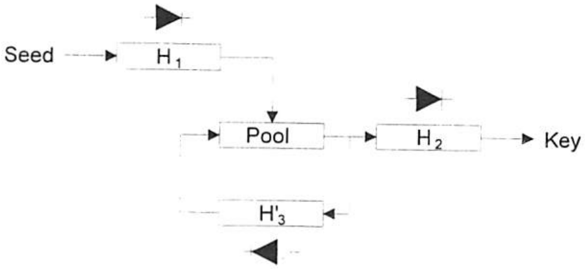


Figure 5: Equivalence to the X9.17 Generator

H_1 mixes the input and prevents chosen-input attacks, H_2 acts as a one-way function for the encryption key, and H'_3 acts as a one-way function for the internal state. This generator is therefore functionally similar to the X9.17 one, but contains significantly more internal state and does not require the use of a rather slow and unexportable triple DES implementation and the secure storage of an encryption key.

4. Polling for Randomness

Now that we have the basic pool management code, we need to fill the pool with random data. To do this we use two methods, a fast randomness poll which executes very quickly and gathers as much random (or apparently random) information as quickly as possible, and a slow poll which can take a lot longer than the fast poll but which performs a more in-depth search for sources of random data. The data sources we use for the generator are chosen to be reasonably safe from external manipulation, since an attacker who tries to modify them to provide predictable input to the generator will either require superuser privileges (which would allow them to bypass any security anyway) or would crash the system when they change operating system data structures.

The sources used by the fast poll are fairly consistent across systems and typically involve obtaining constantly-changing information covering mouse, keyboard, and window states, system timers, thread, process, memory, disk, and network usage details, and assorted other paraphernalia maintained and updated by most operating systems. A fast poll completes very quickly, and gathers a reasonable amount of random information. Scattering these polls throughout the application which will eventually use the random data (in the form of keys or other security-related objects) is a good move, or alternatively they can be embedded inside other functions in a security module so that even careless programmers will (unknowingly) perform fast polls at some point. No-one will ever notice that their RSA signature check takes a few extra microseconds due to the embedded fast poll, and although the presence of the more thorough slow polls may make it slightly superfluous, performing a number of effectively-free fast polls can never hurt.

There are two general variants of the slower randomness-polling mechanism, with individual operating system-specific implementations falling into one of the two groups. The first variant is used with operating systems which provide a rather limited amount of useful information, which tends to coincide with less sophisticated systems which have little or no memory protection and have difficulty performing the polling as a background task or thread. These systems include Win16 (Windows 3.x), the Macintosh, and (to some extent) OS/2, in which the slow randomness poll involves walking through global and system data structures recording information such as handles, virtual addresses, data item sizes, and the large amount of other information typically found in these data structures.

Of the three examples, Win16 provides the most information since it makes all system and process data structures visible to the user through the ToolHelp library, which means we can walk down the list of global heap entries, system modules and tasks, and other data structures. Since even a moderately loaded system can contain over 500 heap objects and 50 modules, we need to limit the duration of the poll to a second or two, which is enough to get information on several hundred objects without halting the calling program for an unacceptable amount of time (and under Win16 the poll will indeed lock up the machine until it completes).

Similarly on the Macintosh we can walk through the list of graphics devices, processes, drivers, and filesystem queues to obtain our information. Since there are typically only a few dozen of these, there is no need to worry about time limits. Under OS/2 there is almost no information available, so even though the operating system provides the capability to do so, there is little to be gained by performing the poll in the background. Unfortunately this lack of random data also provides us with less information than that provided by Win16.

The second variant of the slow polling process is used with operating systems which protect their system and global data structures from general access, but which provide a large amount of other information in the form of system, network, and general usage statistics, and which also allow background polling which means we can take as long as we like (within reasonable limits) to obtain the information we require. These systems include Win32 (Windows 95 and Windows NT) and Unix/BeOS.

The Win32 polling process has two special cases, a Win'95 version which uses the ToolHelp32 functions which don't exist under current versions of NT, and an NT version which uses the NetAPI32 functions and performance data information which doesn't exist under Win'95. In order for the same code to run under both systems, we need to dynamically link in the appropriate routines at runtime using `GetModuleHandle()` or `LoadLibrary()` or the program won't load under one or both of the environments.

Once we have the necessary functions linked in, we can obtain the data we require from the system. Under Win'95 the ToolHelp32 functions provide more or less the same functionality as the Win16 ones (with a few extras added for Win32), which means we can walk through the local heap, all processes and threads in the system, and all loaded modules. A typical poll on a moderately-loaded machine nets 5–15kB of data (not all of which is random or useful, of course).

Under NT the process is slightly different because it currently lacks ToolHelp functionality. Instead, NT keeps track of network statistics using the NetAPI32 library, and system performance statistics by mapping them into keys in the Windows registry. The network information is obtained by checking whether the machine is a workstation or server and then reading network statistics from the appropriate network service. This typically yields around 200 bytes of information covering all kinds of network traffic statistics.

The system information is obtained by reading the system performance data, which is maintained internally by NT and copied to locations in the registry when a special registry key is opened. This creates a snapshot of the system performance at the time the key was opened, and covers a large amount of system information such as process and thread statistics, memory information, disk access and paging statistics, and a large amount of other similar information:

```
RegQueryValueEx( HKEY_PERFORMANCE_DATA,
    "Global", NULL, NULL, buffer, &length );
addToPool( buffer, length );
```

A typical poll on a moderately-loaded machine nets around 30–40kB of data (again, not all of this is random or useful).

The Unix randomness polling is the most complicated of all. Unix systems don't maintain any easily-accessible collections of system information or statistics, and even sources which are accessible with some difficulty (for example kernel data structures) are accessible only to the superuser. However there is a way to access this information which works for any user on the system. Unfortunately it isn't very simple.

Unix systems provide a large collection of utilities which can be used to obtain statistics and information on the system. By taking the output from each of these utilities and adding them to the randomness pool, we can obtain the same effect as using ToolHelp under Win'95 or reading performance information from the registry under NT. The general idea is to identify each of these randomness sources (for example netstat -in) and somehow obtain their output data. A suitable source should have the following three properties:

1. The output should (obviously) be reasonably random.
2. The output should be produced in a reasonable time frame and in a format which makes it suitable for our purposes (an example of an unsuitable source is top, which displays its output interactively). There are often program arguments which can be used to expedite the arrival of data in a timely manner, for example we can tell netstat not to try to resolve host names but instead to produce its output with IP addresses to identify machines.
3. The source should produce a reasonable quantity of output (an example of a source which can produce far too much output is pstat -f, which weighed in with 600kB of output on a large Oracle server. The only useful effect this had was to change the output of vmstat, another useful randomness source).

Now that we know where to get the information, we need to figure out how to get it into the randomness pool. This is done by opening a pipe to the requested source and reading from it until the source has finished producing output. To obtain input from multiple sources, we walk through the list of sources calling `popen()` for each one, add the descriptors to an `fd_set`, make the input from each source non-blocking, and then use `select()` to wait for output to become available on one of the descriptors (this adds further randomness because the fragments of output from the different sources are mixed up in a somewhat arbitrary order which depends on the order and manner in which the sources produce output). Once the source has finished producing output, we close the pipe:

```
for( all potential data sources )
  {
  if( access( source.path, X_OK ) )
    {
    /* Source exists, open a pipe to it */
    source.pipe = popen( source );
    fcntl( source.pipeFD, F_SETFL, O_NONBLOCK
    );
    FD_SET( source.pipeFD, &fds );

    skip all alternative forms of this source
    (eg /bin/pstat vs /etc/pstat);
    }
  }

while( sources are present and
       buffer != full )
  {
  /* Wait for data to become available */
  if( select( ..., &fds, ... ) == -1 )
    break;

  foreach source
    {
    if( FD_ISSET( source.pipeFD, &fds ) )
      {
      count = fread(buffer, source.pipe );
      if( count )
        add buffer to pool;
      else
        pclose( source );
      }
    }
  }
```

Because many of the sources produce output which is formatted for human readability, the code to read the output includes a simple run-length compressor which reduces formatting data such as repeated spaces to the count of the number of repeated characters, conserving space in the data buffer.

Since this information is supposed to be used for security-related applications, we should take a few security precautions when we do our polling. Firstly, we use `popen()` with hard-coded absolute paths instead of simply `exec()`'ing the program used to provide the information. In addition we set our uid to 'nobody' to ensure we can't accidentally read any privileged information if the polling process is running with superuser privileges, and to generally reduce the potential for damage. To protect against very slow (or blocked) sources holding up the polling process, we include a timer which kills a source if it takes too long to provide output. The polling mechanism also includes a number of other safety features to protect against various potential problems, which have been omitted from the pseudocode for clarity.

Because the paths are hard-coded, we may need to look in different locations to find the programs we require. We do this by maintaining a list of possible locations for the programs and walking down it using `access()` to check the availability of the source.

Once we locate the program, we run it and move on to the next source. This also allows us to take into account system-specific variations of the arguments required by some programs by placing the system-specific version of the command to invoke the program first on the affected system (for example IRIX uses a slightly nonstandard argument for the last command, so on SGI systems we try to execute this in preference to the more usual invocation of last).

Due to the fact that popen() is broken on some systems (SunOS doesn't record the pid of the child process, so it can reap the wrong child, resulting in pclose() hanging when it's called on that child), we also need to write our own version of popen() and pclose(), which conveniently allows us to create a custom popen() which is tuned for use by the randomness-gathering process.

Finally, we need to take into account the fact that some of the sources can produce a lot of relatively nonrandom output, the 600kB of pstat output being an extreme example. Since the output is read into a buffer with a fixed maximum size (a block of shared memory as explained in "Extensions to the basic polling model" below), we want to avoid flooding the buffer with useless output. By ordering the sources in the order of usefulness, we can ensure that information from the most useful sources is added preferentially. For example vmstat -s would go before df which would in turn precede arp -a. This ordering also means that late-starting sources like uptime will produce better output when the processor load suddenly shoots up into double digits due to all the other polling processes being forked by the popen().

A typical poll on a moderately-loaded machine nets around 20–40kB of data (with the usual caveat about usefulness).

The slow poll can also check for and use various other sources which might only be available on certain systems. For example some systems have /dev/random drivers which accumulate random event data from the kernel, and some may be fitted with special hardware for generating cryptographically strong random numbers. The slow poll can check for the presence of these sources and use them in preference to or in addition to the usual sources.

Finally, we provide a means to inject externally-obtained randomness into the pool in case other sources are available. A typical external source of randomness would be the user password which, although not random, represents a value which should be unknown to outsiders. Other typical sources include keystroke timings (if the system allows this), the hash of the message being encrypted (another constant but hopefully unknown-to-outsiders quantity), and any other randomness source which might be available. Because of the presence of the mixing function, it's not possible to use this facility to cause any problems with the randomness pool — at worst it won't add any extra randomness, but it's not possible to use it to negatively affect the data in the pool by (say) injecting a large quantity of constant data.

5. Randomness Polling Results

Designing an automated process which is suited to estimating the amount of entropy gathered is a difficult task. Many of the sources are time-varying (so that successive polls will always produce different results), some produce variable-length output (causing output from other sources to change position in the polled data), and some take variable amounts of time to produce data (so that their output may appear before or after the output from faster or slower sources in successive polls). In addition many analysis techniques can be prohibitively expensive in terms of the CPU time and memory required, so we perform the analysis offline using data gathered from a number of randomness sampling runs rather than trying to analyse the data as it is collected.

The field of data compression provides us with a number of analysis tools which can be used to provide reasonable estimates of the change in entropy from one poll to another. The tools we apply to this task are an LZ77 dictionary compressor (which looks for portions of the current data which match previously-seen data) and a powerful statistical compressor (which estimates the probability of occurrence of a symbol based on previously-seen symbols) [23].

The LZ77 compressor uses a 32kB window, which means that any blocks of data already encountered within the last 32kB will be recognised as duplicates and discarded. Since none of the polls generally produce more than 32kB of output, this is adequate for solving the problem of sources which produce variable-length output and sources which take a variable amount of time to produce any output — no matter where the data is located, repeated occurrences will be identified and removed.

The statistical compressor used is an order-1 arithmetic coder, which tries to estimate the probability of occurrence of a symbol based on previous occurrences of that symbol and the symbol preceding it. For example although the probability of occurrence of the letter 'u' in English text is around 2%, the probability of occurrence if the previous letter was a 'q' is almost unity (the exception being words like 'Iraq' and 'Compaq'). The order-1 model therefore provides an tool for identifying any further redundancy which isn't removed by the LZ77 compressor.

By running the compressor over repeated samples, it is possible to obtain a reasonable estimate of how much new entropy is added by successive polls. The use of a compressor to estimate the amount of randomness present in a string leads back to the field of Kolmogorov-Chaitin complexity, which defines a random string as one which has no shorter description than itself, so that it is incompressible. The compression process therefore provides an estimate of the amount of nonrandomness present in the string. A similar principle is used in Maurers universal statistical test for random bit generators, which employs a bitwise LZ77 algorithm to estimate the amount of randomness present in a bit string [24].

The test results were taken from a number of systems and cover Windows 3.x, Windows'95, Windows NT, and Unix systems running under both light and moderate to heavy loads. In addition a reference data set, which consisted of a set of text files derived from a single file, with a few lines swapped and a few characters altered in each file, was used to test the entropy estimation process.

In every case a number of samples were gathered and the change in compressibility relative to previous samples taken under both identical and different conditions was checked. As more samples were processed by the compressor, it adapted itself to the characteristics of each sample and so produced better and better compression (that is, smaller and smaller changes in compression) for successive samples, settling down after the second or third sample. The exception was the test file, where the compression jumped from 55% on the first sample to 97% for all successive samples due to the similarity of the data (the reason it didn't go to over 99% was because of the way the compressor encodes the lengths of repeated data blocks. For virtually all normal data there are many matches for short to medium-length blocks and almost no matches for long blocks, so the compressor's encoding is tuned to be efficient in this range and it emits a series of short to medium length matches instead of a single very long length of the type present in the test file. This means the absolute compressibility is less than it is for the other data, but since our interest is the change in compressibility from one sample to another this doesn't matter much).

The behaviour for the test file indicates that the compressor provides a good tool for estimating the change in entropy — after the first test sample has been processed, the compressed size changes by only a few bytes in successive samples, so the compressor is doing a good job of identifying data which remains unchanged from one sample to the next.

The fast polls, which gather very small amounts of constantly-changing data such as high-speed system counters and timers and rapidly-changing system information, aren't open to automated analysis using the compressor, both because they produce different results on each poll (even if the results are relatively predictable), and because the small amount of data gathered leaves little scope for effective compression. Because of this, only the more thorough slow polls which gather large amounts of information were analysed. The fast polls can be analysed if necessary, but vary greatly from system to system and require manual scrutiny of the sources used rather than automated analysis.

The Win16/Win32 systems were tested both in the unloaded state with no applications running, and in the moderately/heavily loaded state with MS Word, Netscape, and MS Access running. It is interesting to note that even the (supposedly unloaded) Win32 systems had around 20 processes and 100 threads running, and adding the three "heavy load" applications added (apart from the 3 processes) only 10-15 threads (depending on the system). This indicates that even on a supposedly unloaded Win32 system, there is a fair amount of background activity going on (for example both Netscape and MS Access can sometimes consume 100% of the free CPU time on a system, in effect taking over the task of the idle process which grinds to a halt while they are loaded but apparently inactive).

The first set of samples we discuss are the ones which came from the Windows 3.x and Windows'95 systems, and which were obtained using the ToolHelp/ToolHelp32 functions which provide a record of the current system state. Since the results for the two systems were relatively similar, only the

Windows'95 ones will be discussed here. In most cases the results were rather disappointing, with the input being compressible by more than 99% once a few samples had been taken (since the data being compressed wasn't pathological test data, the compression match-length limit described above for the test data didn't occur). The tests run on a minimally-configured machine (one floppy drive, hard drive, and CDROM drive) produced only about half as much output as tests run on a maximally-configured machine (one floppy drive, two hard drives, network card, CDROM drive, SCSI hard drive and CDROM writer, scanner, and printer), but in both cases the compressibility had reached a constant level by the third sample (in the case of the minimal system it reached this level by the second sample). Furthermore, results from polls run one after the other showed little change to polls which were spaced at 1 minute intervals to allow a little more time for the system state to change.

The one very surprising result was the behaviour after the machine was rebooted, with samples taken in the unloaded state as soon as all disk activity had finished. In theory the results should have been very poor because the machine should be in a pristine, relatively fixed state after each reboot, but instead the compressed data was 2½ times larger than it had been when the machine had been running for some time. This is because the plethora of drivers, devices, support modules, and other paraphernalia which the system loads and runs at boot time (all of which vary in their behaviour and performance and in some cases are loaded and run in nondeterministic order) perturb the characteristics sufficiently to provide a relatively high degree of entropy after a reboot. This means that the system state after a reboot is relatively unpredictable, so that although multiple samples taken during one session provide relatively little variation in data, samples taken between reboots do provide a fair degree of variation.

This hypothesis was tested by priming the compressor using samples taken over a number of reboots and then checking the compressibility of a sample taken after the system had been running for some time relative to the samples taken after the reboot. In all cases the compressed data was 4 times larger than it had been when the compressor was primed with samples taken during the same session, which confirmed the fact that a reboot creates a considerable change in system state. This is an almost ideal situation when the data being sampled is used for cryptographic random number generation, since an attacker who later obtains access to the machine used to generate the numbers has less chance of being able to determine the system state at the time the numbers were generated (provided the machine has been rebooted since then).

The next set of samples came from Windows NT systems and record the current network statistics and system performance information. Because of its very nature, it provides far more variation than the data collected on the Windows 3.x/Windows'95 systems, with the data coming from a dual-processor P6 server in turn providing more variation than the data from a single-processor P5 workstation. In all cases the network statistics provide a disappointing amount of information, with the 200-odd bytes collected compressing down to a mere 9 bytes by the time the third sample is taken. Even rebooting the machine didn't help much. Looking at the data collected revealed that the only things which changed much were one or two packet counters, so that virtually all the entropy provided in the samples comes from these sources.

The system statistics were more interesting. Whereas the Windows 3.x/Windows'95 polling process samples the absolute system state, the NT polling samples the change in system state over time, and it would be expected that this time-varying data would be less compressible. This was indeed the case, with the data from the server only compressible by about 80% even after multiple samples were taken (compared to 99^{+}% for the non-NT machines). Unlike the non-NT machines though, the current system loading did affect the results, with a completely unloaded machine producing compressed output which was around 1/10 the size of that produced on the same machine with a heavy load, even though the original, uncompressed data quantity was almost the same in both cases. This is because, with no software running, there is little to affect the statistics kept by the system (no disk or network access, no screen activity, and virtually nothing except the idle process active). Attempting to further influence the statistics (for example by having several copies of Netscape trying to load data in the background) produced almost no change over the canonical "heavy load" behaviour.

The behaviour of the NT machines after being rebooted was tested in a manner identical to the tests which had been applied to the non-NT machines. Since NT exhibits differences in behaviour between loaded and unloaded machines, the state-after-reboot was compared to the state with applications running rather than the completely unloaded state (corresponding to the situation where the user has rebooted their machine and immediately starts a browser or mailer or other program which requires random numbers). Unlike the non-NT systems, the data was slightly more compressible relative to the samples taken immediately after the reboot (which means it compressed by about 83% instead of 80%). This is possibly because the relative change from an initial state to the heavy-load state is less than the change from one heavy-load state to another heavy-load state.

The final set of samples came from a variety of Unix systems ranging from a relatively lightly-loaded Solaris machine to a heavily-loaded multiprocessor student Alpha. The randomness output varied greatly between machines and depended not only on the current system load and user activity but also on how many of the required randomness sources were available (many of the sources are BSD-derived, so systems which lean more towards SYSV, like the SGI machines which were tested, had less randomness sources available than BSD-ish systems like the Alpha).

The results were fairly mixed and difficult to generalise. Like the NT systems, the Unix sources mostly provide information on the change in system state over time rather than absolute system state, so the output is inherently less compressible than it would be for sources which provide absolute system state information. The use of the run-length coder to optimise use of the shared memory buffer further reduces compressibility, with the overall compressibility between samples varying from 70–90% depending on the system.

Self-preservation considerations prevented the author from exploring the effects of rebooting the multiuser Unix machines.

6. Extensions to the Basic Polling Model

On a number of systems we can hide the lengthy slow poll by running it in the background while the main program continues execution. As long as the slow poll is started a reasonable amount of time before the random data is needed, the slow polling will be invisible to the user. In practice by starting the poll as soon as the program is run, and having it run in the background while the user is connecting to a site or typing in their password or whatever else the program requires, the random data is available when it is required.

The background polling is run as a thread under Win32 and as a child process under Unix. Under Unix the polling information is communicated back to the parent process using a block of shared memory, under Win32 the thread shares access to the randomness pool with the other threads in the process which makes the use of explicitly shared memory unnecessary. The general method used to prevent simultaneous access to the pool is simply that if a background poll is in progress we wait for it to run to completion before allowing the access. The code to extract data from the pool then becomes:

```
extractData()
  {
  if( no random data available and no
      background poll in progress )
     /* Caller forgot to perform slow poll */
     start a slow poll;

  wait for any background poll to run to
      completion;
  if( still no random data available )
     error;

  extract/mix data from the pool;
  }
```

In fact under Win32 we can provide a much finer level of control than this somewhat crude "don't allow any access if a poll is in progress" method. By using semaphores we can control access to the pool so that the fact that a background poll is active doesn't stop us from using the pool at the same time. This is done by using Win32 "critical sections" (which aren't really critical sections at all, but a form of fast mutex which are used to stop more than one thread from holding a resource at any one time). By wrapping up access to the random pool in a mutex, we can allow a background poll to independently update the pool in between reading data from it. The previous pseudocode can be changed to make it thread-safe by changing the last few lines to:

```
EnterCriticalSection( ... );
extract/mix data from the pool;
LeaveCriticalSection( ... );
```

The background polling thread also contains these calls, which ensures that only one thread will try to access the pool at a time. If another thread tries to access the pool, it is suspended until the thread which is currently accessing the pool has released the mutex, which happens extremely quickly since the only operation being performed is either a mixing operation or a copying of data.

Now that we have a nice, thread-safe means of performing more or less transparent updates on the pool, we can extend the basic manually-controlled polling model even further for extra user convenience. The first two lines of the extractData() pseudocode contain code to force a slow poll if the calling application has forgotten to do this (the fact that the application grinds to a halt for several seconds will hopefully make this mistake obvious to the programmer the first time they test their application). We can make the polling process even more foolproof by performing it automatically ourselves without programmer intervention. As soon as the security or randomness subsystem is started, we begin performing a background slow poll, which means the random data becomes available as soon as possible after the application is started (this also requires a small modification to the function which manually starts a slow poll so that it won't start a redundant background poll if the automatic poll is already taking place).

In general an application will fall into one of two categories, either a client-type application such as a mail reader or browser which a user will start up, perform one or more transactions or operations with, and then close down again, and a server-type application which will run over an extended period of time. In order to take both of these cases into account, we perform one poll every minute for the first 5 minutes to quickly obtain random data for active client-type applications, and then drop back to one poll every 10 minutes for longer-running server-type applications (this is also useful for client applications which are left to run in the background, mail readers being a good example).

7. Protecting the Randomness Pool

The randomness pool presents an extremely valuable resource, since any attacker who gains access to it can use it to predict any private keys, encryption session keys, and other valuable data generated on the system. Using the design philosophy of "Put all your eggs in one basket and watch that basket very carefully", we go to some lengths to protect the contents of the randomness pool from outsiders. Some of the more obvious ways to get at the pool are to recover it from the page file if it gets swapped to disk, and to walk down the chain of allocated memory blocks looking for one which matches the characteristics of the pool. Less obvious ways are to use sophisticated methods to recover the contents of the memory which contained the pool after power is removed from the system.

The first problem to address is that of the pool being paged to disk. Fortunately several operating systems provide facilities to lock pages into memory, although there are often restrictions on what can be achieved. For example many Unix versions provide the mlock() call, Win32 has VirtualLock() (which, however, is implemented as { return TRUE; } under Windows 95), and the Macintosh has HoldMemory(). A discussion of various issues related to locking memory pages (and the difficulty of erasing data once it has been paged to disk) is given in Gutmann [25].

If no facility for locking pages exists, the contents can still be kept out of the common swap file through the use of memory-mapped files. A newly-created memory-mapped file can be used as a private swap file which can be erased when the memory is freed (although there are some limitations on how well the data can be erased — again, see Gutmann [25]). Further precautions can be taken to make the private swap file more secure, for example the file should be opened for exclusive use and/or have the strictest possible access permissions, and file buffering should be disabled if possible to avoid the buffers being swapped (under Win32 this can be done by using the FILE_FLAG_NO_BUFFERING flag when calling CreateFile(); some Unix versions have obscure ioctl's which achieve the same effect).

The second problem is that of another process scanning through the allocated memory blocks looking for the randomness pool. This is aggravated by the fact that, if the randomness-polling is built into an encryption subsystem, the pool will often be allocated and initialised as soon as the security subsystem is started, especially if automatic background polling is used.

Because of this, the memory containing the pool is often allocated at the head of the list of allocated blocks, making it relatively easy to locate. For example under Win32 the VirtualQueryEx() function can be used to query the status of memory regions owned by other processes, VirtualUnprotectEx() can be used to remove any protection, and ReadProcessMemory() can be used to recover the contents of the pool or, for an active attack, set its contents to zero. Generating encryption keys from a buffer filled with zeroes (or the hash of a buffer full of zeroes) can be hazardous to security.

Although there is no magic solution to this problem, the task of an attacker can be made considerably more difficult by taking special precautions to obscure the identity of the memory being used to implement the pool. This can be done both by obscuring the characteristics of the pool (by embedding it in a larger allocated block of memory containing other data) and by changing its location periodically (by allocating a new memory block and moving the contents of the pool to the new block). The relocation of the data in the pool also means it is never stored in one place long enough to be retained by the memory it is being stored in, making it harder for an attacker to recover the pool contents from memory after power is removed [25].

This obfuscation process is a simple extension of the background polling process. Every time a poll is performed, the pool is moved to a new, random-sized memory block and the old memory block is wiped and freed. In addition, the surrounding memory is filled with non-random data to make a search based on match criteria of a single small block filled with high-entropy data more difficult to perform (that is, for a pool of size *n* bytes, a block of *m* bytes is allocated and the *n* bytes of pool data are located somewhere within the larger block, surrounded by *m-n* bytes of other data). This means that as the program runs, the pool becomes buried in the mass of memory blocks allocated and freed by typical GUI-based applications. This is especially apparent when used with frameworks such as MFC, whose large (and leaky) collection of more or less arbitrary allocated blocks provides a perfect cover for a small pool of randomness.

Since the obfuscation is performed as a background task, the cost of moving the data around is almost zero. The only time when the randomness state is locked (and therefore inaccessible to the program) is when the data is being copied from the old pool to the new one:

```
allocate new pool;
write nonrandom data to surrounding memory;
lock randomness state (EnterCriticalSection()
     under Win32);
copy data from old pool to new pool;
unlock randomness state
     (LeaveCriticalSection() under Win32);
zeroise old pool;
```

This assumes that operations which access the randomness pool are atomic and that no portion of the code will try to retain a pointer to the pool between pool accesses.

We can also use this background thread or process to try to prevent the randomness pool from being swapped to disk. The reason this is necessary is that the techniques suggested previously for locking memory aren't completely reliable: `mlock()` can only be called by the superuser, `VirtualLock()` doesn't do anything under Windows'95, and even under Windows NT where it is actually implemented, it doesn't do what the documentation says. Instead of making the memory completely non-swappable, it is only kept non-swappable as long as at least one thread in the process which owns the memory is active. Once all threads are pre-empted, the memory can be swapped to disk just like non-"locked" memory [26]. Although the precise behaviour of `VirtualLock()` isn't known, it appears that it acts as a form of advisory lock which tells the operating system to keep the pages resident for as long as possible before swapping them out.

Since the correct functioning of the memory-locking facilities provided by the system can't be relied upon, we need to provide an alternative method to try to retain the pages in memory. The easiest way to do this is to use the background thread which is being used to relocate the pool to continually touch the pages, thus ensuring they are kept at the top of the swappers LRU queue. We do this by decreasing the sleep time of the thread so that it runs more often, and keeping track of how many times we have run so that we only relocate the pool as often as the previous, less-frequently-active thread did:

```
touch randomness pool;
if( time to move the pool )
  {
  move the pool;
  reset timer;
  }
sleep;
```

This is especially important when the process using the pool is idle over extended periods of time, since pages owned by other processes will be given preference over those of the process owning the pool. Although the pages can still be swapped when the system is under heavy load, the constant touching of the pages makes it less likely that this swapping will occur under normal conditions.

8. Conclusion

The random number generator described in this paper has proven to be relatively portable across different systems (a generator similar to the one described here has been implemented in the cryptlib encryption library [27] which has been in use on a wide variety of systems for around 2 years), provides a good source of practically strong random data on most systems, and can be set up to function independently of special hardware or the need for user or programmer input, which is often not available.

Acknowledgments

The author would like to thank Peter Fenwick, John Kelsey, Colin Plumb, Robert Rothenburg Walking-Owl, and the referees for their advice and input during the preparation of this paper.

References

[1] "Cryptographic Randomness from Air Turbulence in Disk Drives", Don Davis, Ross Ihaka, and Philip Fenstermacher, Proceedings of Crypto '94, Springer-Verlag Lecture Notes in Computer Science, No.839, 1994.

[2] "Truly Random Numbers", Colin Plumb, Dr.Dobbs Journal, November 1994, p.113.

[3] "PGP Source Code and Internals", Philip Zimmermann, MIT Press, 1995.

[4] "Random noise from disk drives", Rich Salz, posting to cypherpunks mailing list, message-ID `9601230431.AA06742@sulphur.osf.org` , 22 January 1996.

[5] "My favourite random-numbers-in-software package (unix)", Matt Blaze, posting to cypherpunks mailing list, message-ID `199509301946.PAA15565@crypto.com`, 30 September 1995.

[6] "Using and Creating Cryptographic-Quality Random Numbers", John Callas, `http://www.merrymeet.com/-jon/usingrandom.html`, 3 June 1996.

[7] "Suggestions for random number generation in software", Tim Matthews, RSA Data Security Engineering Report, 15 December 1995.

[8] "Applied Cryptography (Second Edition)", Bruce Schneier, John Wiley and Sons, 1996.

[9] "Cryptographic Random Numbers", IEEE P1363 Working Draft, Appendix G, 6 February 1997.

[10] "Randomness Recommendations for Security", Donald Eastlake, Stephen Crocker, and Jeffrey Schiller, RFC 1750, December 1994.

[11] "The Art of Computer Programming: Volume 2, Seminumerical Algorithms", Donald Knuth, Addison-Wesley, 1981.

[12] "Handbook of Applied Cryptography", Alfred Menezes, Paul van Oorschot, and Scott Vanstone, CRC Press, 1996.

[13] "Foundations of Cryptography — Fragments of a Book", Oded Goldreich, February 1995, `http://theory.lcs.mit.edu/-~oded/frag.html`.

[14] "Netscape's Internet Software Contains Flaw That Jeopardizes Security of Data", Jared Sandberg, The Wall Street Journal, 18 September 1995.

[15] "Randomness and the Netscape Browser", Ian Goldberg and David Wagner, Dr.Dobbs Journal, January 1996.

[16] "Breakable session keys in Kerberos v4", Nelson Minar, posting to the cypherpunks mailing list, message-ID `199602200828.BAA21074@-nelson.santafe.edu`, 20 February 1996.

[17] "X Authentication Vulnerability", CERT Vendor-Initiated Bulletin VB-95:08, 2 November 1995.

[18] "Proper Initialisation for the BSAFE Random Number Generator", Robert Baldwin, RSA Laboratories' Bulletin, 25 January 1996.

[19] "How to Attack a PRNG", John Kelsey, Bruce Schneier, David Wagner, and Chris Hall, to appear.

[20] "SFS — Secure FileSystem", Peter Gutmann, `http://www.cs.auckland.ac.nz/-~pgut001/sfs.html`.

[21] `/dev/random` driver source code (random.c), Theodore T'so, 24 April 1996.

[22] "American National Standard for Financial Institution Key Management (Wholesale)", American Bankers Association, 1985.

[23] "Practical Dictionary/Arithmetic Data Compression Synthesis", Peter Gutmann, MSc thesis, University of Auckland, 1992.

[24] "A Universal Statistical Test for Random Bit Generators", Ueli Maurer, Proceedings of Crypto '90, Springer-Verlag Lecture Notes in Computer Science, No.537, 1991, p.409.

[25] "Secure deletion of data from magnetic and solid-state memory", Peter Gutmann, Sixth Usenix Security Symposium proceedings, July 22-25, 1996, San Jose, California.

[26] "Advanced Windows (third edition)", Jeffrey Richter, Microsoft Press, 1997.

[27] "cryptlib Free Encryption Library", Peter Gutmann, `http://www.cs.auckland.ac.nz/-~pgut001/cryptlib/`.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of developers, programmers, system administrators, and architects working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering innovation and research that works
- communicating rapidly the results of both research and innovation
- providing a neutral forum for the exercise of critical thought and the airing of technical issues

SAGE, the System Administrators Guild

The System Administrators Guild, a Special Technical Group within the USENIX Association, is dedicated to the recognition and advancement of system administration as a profession. To join SAGE, you must be a member of USENIX.

Member Benefits:

- Free subscription to *;login:*, the Association's magazine, published eight times a year, featuring technical articles, system administration tips and techniques, practical columns on Perl, Javan and C++, book and software reviews, summaries of sessions at USENIX conferences, Snitch Reports from the USENIX representative and others on various ANSI, IEEE, and ISO standards efforts.
- Access to papers from the USENIX Conferences and Symposia, starting with 1993, via the USENIX Online Library on the World Wide Web.
- Discounts on registration fees for the annual, multi-topic technical conference, the System Administration Conference (LISA), and the various single-topic symposia addressing topics such as object-oriented technologies, security, operating systems, electronic commerce, and NT - as many as ten technical meetings every year.
- Discounts on the purchase of proceedings and CD-ROMs from USENIX conferences and symposia and other technical publications.
- PGP Key Signing Service (available at conferences).
- Discount on BSDI, Inc. products.
- Discount on the five volume set of 4.4BSD manuals plus CD-ROM published by O'Reilly & Associates, Inc. and USENIX.
- Discount on all publications and software from Prime Time Freeware.
- 20% discount on all titles from O'Reilly & Associates.
- Savings (10-20%) on selected titles from McGraw-Hill, The MIT Press, Morgan Kaufmann Publishers, Sage Science Press, and John Wiley & Sons.
- Special subscription rate for *The Linux Journal* and *The Perl Journal.*
- The right to vote on matters affecting the Association, its bylaws, election of its directors and officers.

Supporting Members of the USENIX Association:

Adobe Systems Inc.
Advanced Resources
ANDATACO
Apunix Computer Services
Auspex Systems, Inc.
Boeing Commercial
Crosswind Technologies, Inc.
Digital Equipment Corporation
Earthlink Network, Inc.
Invincible Technologies
Lucent Technologies, Bell Labs
Motorola Research & Development
MTI Technology Corporation
Nimrod AS
O'Reilly & Associates
Sun Microsystems, Inc.
Tandem Computers, Inc.
UUNET Technologies, Inc.

Sage Supporting Members:

Atlantic Systems Group
Digital Equipment Corporation
ESM Services, Inc.
Global Networking and Computing, Inc.
Great Circle Associates
OnLine Staffing
O'Reilly & Associates
Sprint Paranet
Texas Instruments, Inc.
TransQuest Technologies, Inc.
UNIX Guru Universe

For further information about membership, conferences or publications, contact: USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA. Phone: 510-528-8649. Fax: 510-548-5738. Email: *office@usenix.org.*
URL: *http://www.usenix.org.*

ISBN 1-880446-92-8